More on The Architecture of Kafka

In this lesson, we'll continue learning about the architecture of Kafka.

WE'LL COVER THE FOLLOWING

Commit

For each consumer, **Kafka stores the offset for each partition**. This offset indicates which record in the partition the consumer read and processed last. It helps Kafka to ensure that each record is eventually handled.

When consumers have processed a record, they commit a new offset. In this way, Kafka knows at all times which records have been processed by which consumer and which records still have to be processed. Of course, consumers can commit records before they are actually processed. As a result, **records that never get processed is a possibility**.

The commit is on an offset, for example, “all records up to record 10 in this partition have been processed.” **A consumer can commit a batch of records**, which results in better performance because fewer commits are required.

But then **duplicates can occur**. This happens when the consumer fails after processing a part of a batch and has not yet committed the entire batch. At restart, the application would read the complete batch again, because Kafka

restarts at the last committed record and thus at the beginning of the batch.

Kafka also supports **exactly once semantics** that is, a guaranteed one-time delivery.

Polling

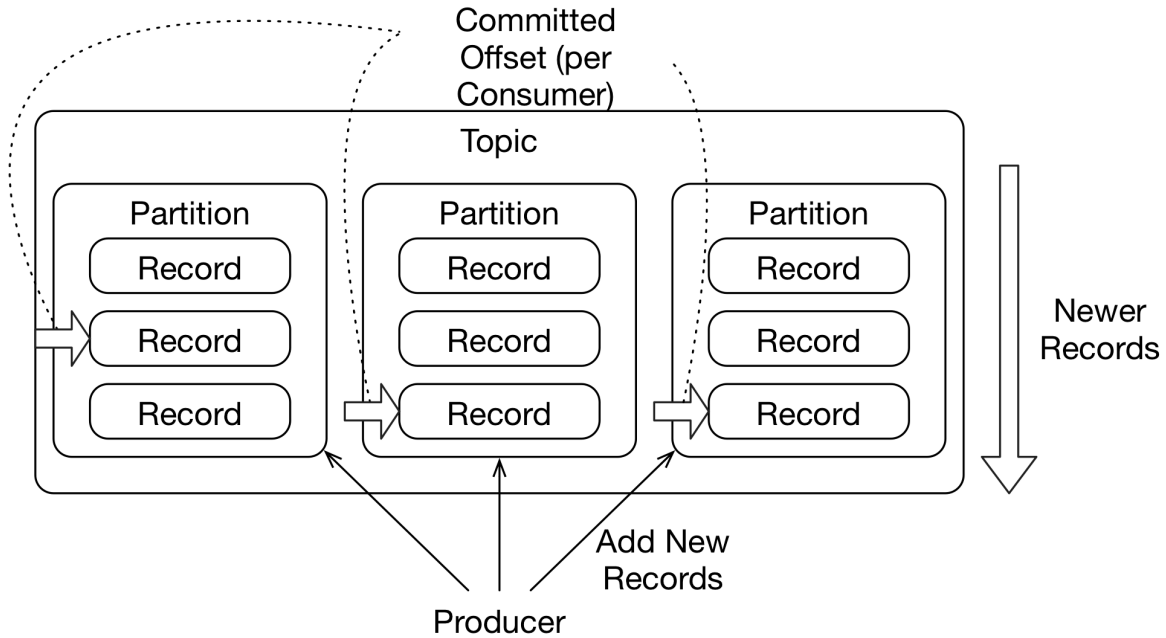
> The consumers **poll** the data, meaning they fetch new data and process it.

With the push model, on the other hand, the producer would send the data to the consumer.

Polling doesn't seem to be very elegant. However, in the absence of a push, the consumers are protected from too much load when a large number of records are being sent and have to be processed. **Consumers can decide for themselves when they process the records**.

Libraries like Spring Kafka for Java, which is used in the example, poll new records in the background. The developer implements methods to handle new records. Spring Kafka then calls them. The polling is hidden from the developer.

Records, topics, partitions, and commits



Partitions and Topics in Kafka

The drawing above shows an example. The topic is divided into **three partitions**, each containing three records.

In the lower part of the figure are the newer records. The producer creates new records at the bottom. The consumer has not yet committed the latest record for the first partition but has for all other partitions.

Replication

Partitions store the data. Because data in one partition is independent of data in the other partitions, **partitions can be distributed over servers**:

- Each server then processes some partitions. This allows load balancing.
- To handle a larger load, new servers need to be added and some partitions need to be moved to the new server.
- The partitions can also be **replicated** so that the data is stored on several servers, meaning Kafka can be made fail-safe. If one server crashes or loses its data, other replicas still exist.

Example

- The number N of replicas can be configured. When writing, you can determine how many in-sync replicas must commit changes.
- With $N = 3$ replicas and two in-sync replicas, the cluster remains available even if one of the three replicas fails.
- Even if one server fails, new records can still be written to two replicas. If a replica fails, no data is lost because every write operation must have been successful on at least two replicas.
- Even if a replica is lost, the data must still be stored on at least one additional replica.

Kafka thus **supports some fine tuning** regarding the CAP theorem (see Events) by changing the number of replicas and in-sync replicas.

Leader and follower

The replication is implemented in such a way that one leader writes and the remaining replicas write as followers. The producer writes directly to the

remaining replicas write as followers. The producer writes directly to the leader. Several write operations can be combined in a batch.

On the one hand, it takes longer before a batch is complete and for the changes to be actually saved. On the other hand, throughput increases because it is more efficient to store multiple records at once.

The overhead of coordinating the writes happens just once for the full batch and not for each record.

Retry sending

If the transfer to the consumer was not successful, the producer can use the API to specify that the transfer is attempted again. The default setting is that sending a record is not repeated, thus causing **records to be lost**. If the transfer is configured to occur more than once, the record may already have been successfully transferred despite the error. In this case, **there would be a duplicate**, which the consumer would have to deal with.

One possibility is to develop the consumer in such a way that it offers **idempotent processing**. This means that the consumer is in the same state, no matter how often the consumer processes a record (see Challenges).

For example, if a duplicate is received, the consumer can determine that it has already modified the record accordingly and ignore it.

QUIZ

1 Suppose a Kafka consumer has committed that it has processed 20 records. However, it fails mid-processing. Which of the following statements best describes what happens next?

In the next lesson, we'll continue our discussion on the architecture of Kafka.