**Kafka**

Kafka is an open source distributed data streaming platform, used for high performance data pipelines, streaming analytics, data integrations and mission critical applications.

**What is Kafka Broker**

Kafka Brokers are separate processes that operate on diff machines and communicate with each other with high-speed, fault tolerant network. It is a server, that acts a as a broker between a producer and a consumer. Consumer and Produces do not communicate directly to each other. They use Kafka Broker as an agent to exchange messages. There are at least 3 Kafka Brokers in a Cluster.

**KRaft/Zookeeper**

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It is used to store metadata information about brokers, topics and partitions. Write to Zookeeper are only performed on change to the membership of consumer groups or changes to the Kafka cluster itself. But today, Zookeeper is deprecated and it is better to use KRaft, it is a new consensus protocol and is event base, which means it’s better suited for scaling.

**Kafka Cluster**

Kafka Cluster is a group of Brokers that act together for three reasons: scalability, durability and speed. Data is replicated across multiple servers, which ensures their availability and consistency. Kafka also balances load between multiple brokers.

**Producer**

Producer is an application that sends messages to a Kafka Broker.

**Consumer**

Consumer is an application, that reads messages from Kafka Broker. So basically, any application, that requests data from a Kafka Broker is a Consumer. And it asks for the data they want and have permission to consume.

**Topics**

Messages in Kafka are categorized as topics. They are like tables in databases or a folder in a filesystem. Consumers send data to topics and consumers request data from it. Topics are identified by names. You can have any number of topics, but currently Kafka can handle hundreds of thousands of topics, depending on partitions.

**Partitions**

Kafka topics are additionally broken down by partitions, which contain records in unchangeable sequence. Kafka messages will store messages in topics. But sometimes this data can be enormous and it may not be possible to store it in a single computer. So Kafka will store it into multiple partitions in a distributed fashion. You need to provide a key to store it in a partition. Kafka will hash the kay and use the module of the hashed value for storing, therefore a data with the same key will be consistently saved.

**Offset**

Offset is a sequence of Ids messages are assigned in offset. Once offset is assigned to a message it will never change. The first message has an offset of zero, next 1 and so on.

**Consumer Groups**

Consumer group is a unity of multiple consumers that are working together to process messages.

**Message**

Kafka message, also known as event or record, it is a key value pair that we send to the broker. There are 4 parameters that we are able to send:

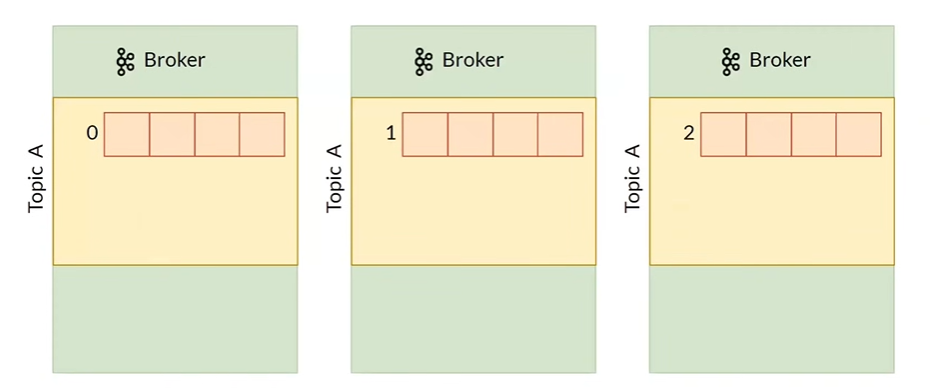
**Key(optional) –** used to store it in the exact location(partition) in the cluster.

**Value –** byte array, this is the data you want to store.

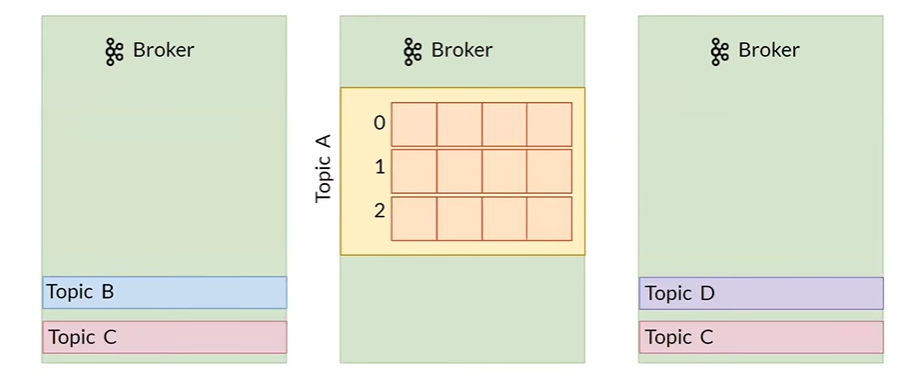
**Timestamp(optional) –** time of the message. If null, Kafka will take care of it.

**Headers –** object of key values of headers.

**Topic Placement by Brokers(in cluster)**



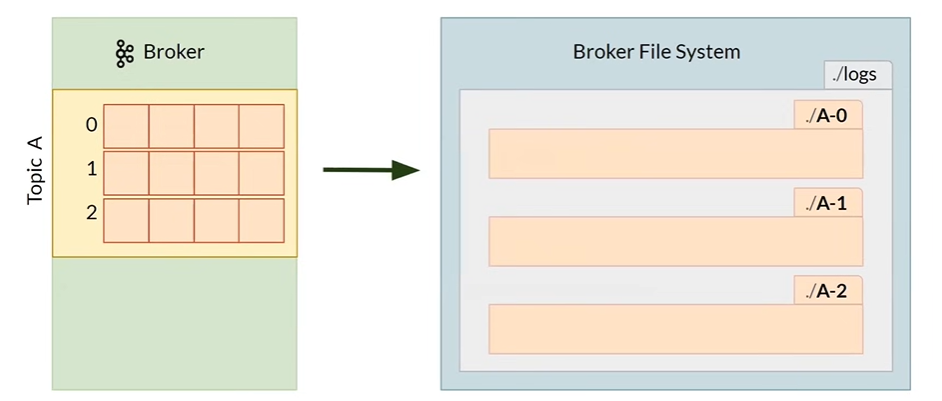
This is the perfect placement of partitions per Broker. But There is one downside, not always will partitions be so well arranged for every topic, because Kafka is configured the way it arranges the total number of partitions for every topic in total and tries to make it even for every broker. For example if there are 4 topics and they have 150 partitions in total, Kafka will give every broker 50 partitions, no matter of which topic they are in, so there is a chance for a topic to have every partition on one broker. Like that:



If such thing happens, you can reconfigure the arrangement of partitions manually.

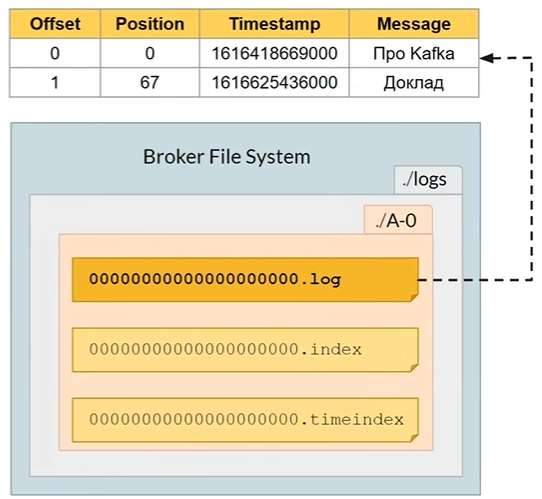
**Where are the events stored?**

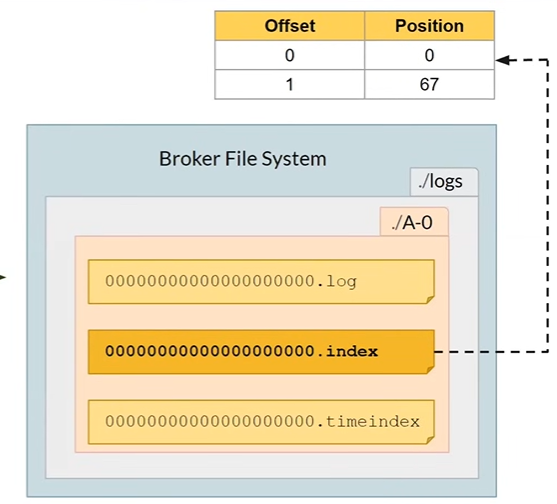
They are stored in log files:



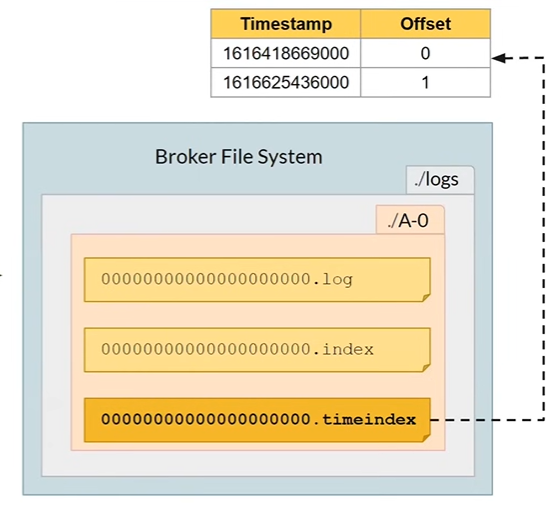
This is folder for a topic A. ./A-0 is a folder for logs of partition 0 and so on.

A folder of a partition looks like this:





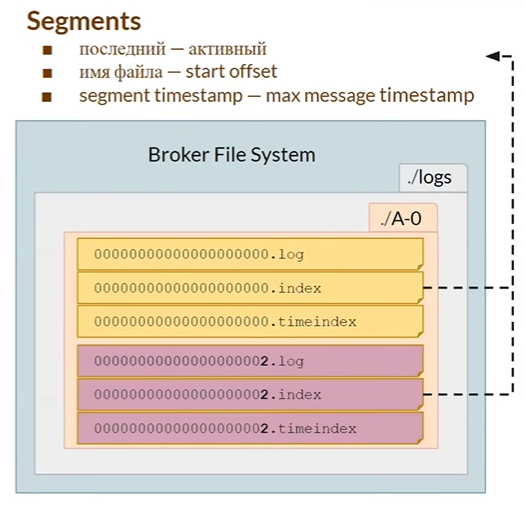
This is the file that we use to get data from an offset we want. For example, if we want to read a log with the offset 1, we check this file and read from 67th byte.



This file is used when we want to read something from particular time. If we use random timestamps there can be any value stored, even a timestamp of future. But if we do not set the timestamp manually, Kafka will take care of it and it can be trusted.

**SEGMENTS**

Log file has a limit(1GB), when 1GB is used the log file freezes and new one is created, with the number of offset of the first log stored. This sections of partitions are called Segments. Only the latest segment can be active. Active segment is defined by its’ name. Segments have timestamp, which is max timestamp of messages in this segment.



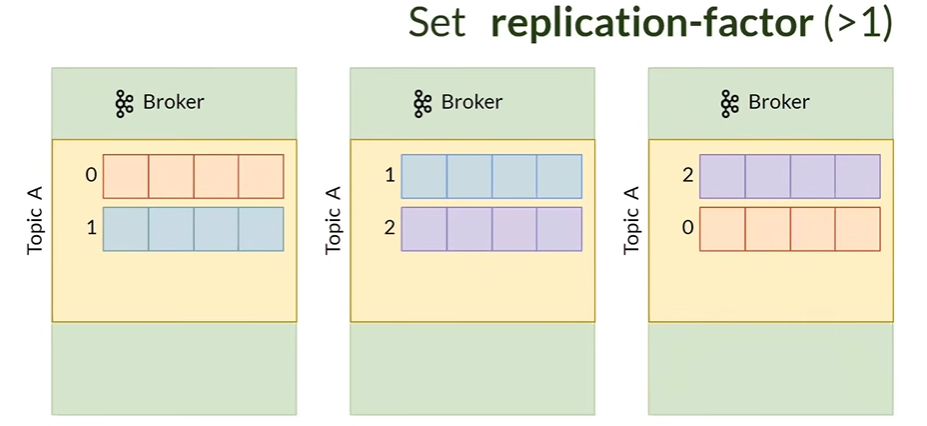
There offset of the first log in partition is 2, so 000…2.log is the name of the active log file. There can be only 1 active log file.

**Deleting data from Kafka Topic**

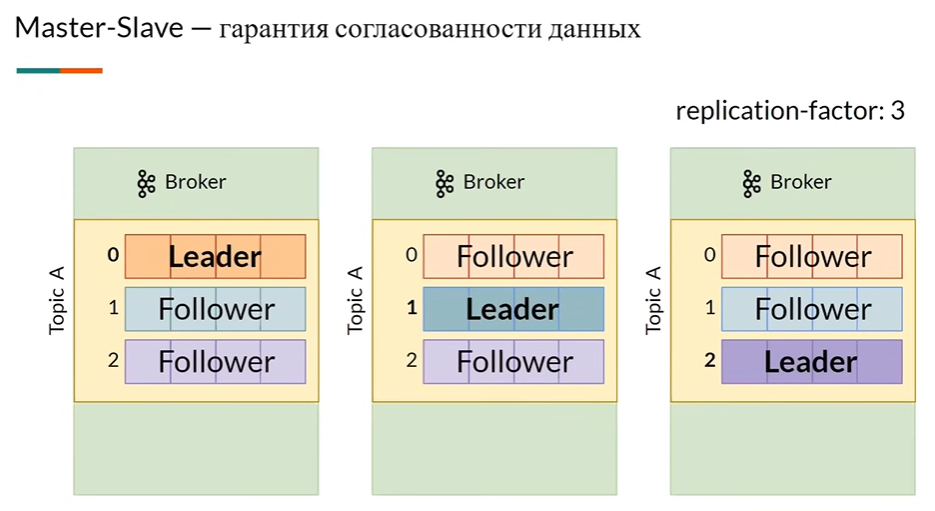
You can not manually delete date in Kafka topic, it is not supported. But there is an automatic removal of data configured by TTL parameter. But whole segments whose timestamp is expired are deleted, not the messages separately. There is one drawback from it, if somebody sets the timestamp of future on a message, then the whole segment won’t be deleted until this message is expired.

**Data Replication(fault tolerance and reliability)**

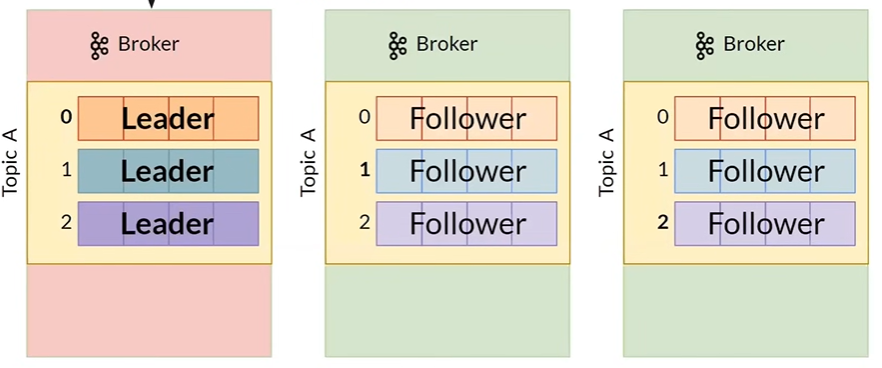
If we have one replica of a partition for the whole cluster, then if the broker where the partition was is down, we will loose every data that was on the partition. There for there is a configuration of replication factor, where you set the number greater then one, to have more then one replica of a partition per cluster. Kafka will take care that replicas of a partition won’t be stored on the same broker.



But data can be inconsistent between this replicas. For fixing it Kafka has Master-Slave architecture for partitions, where one of the partition is chosen as a Leader and others are followers. We always read and write on a leader replica.



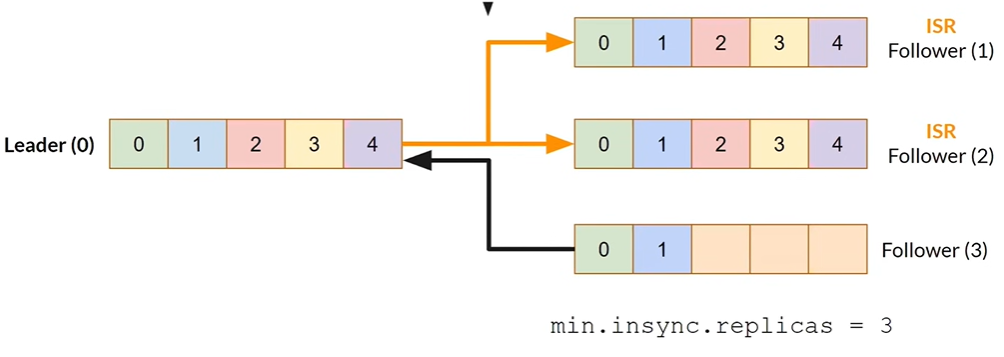
There is one catch, every leader partition can be on the same broker, this means only one broker will be under load and others will be standby.



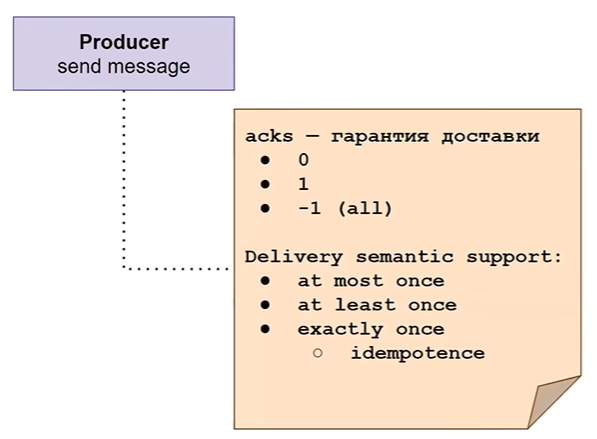
**Data sync between replicas**

Followers must fetch data from leader periodically. To ensure that at least one replica can be used as a leader, if the current leader falls, there are ISR(in-sync replicas). When we write in a leader replica, leader replica performs sync write to an ISR follower, therefore ensures, that they can be used as a leader if something happens to it.

You can configure number of insync replicas. You need to carefully configure this number, because if there are no sufficient number of replicas to match the configured number, write will fail and won’t be performed. In this picture there are 3 insync replicas. 1 Leader and other ISR followers.



**Sending message from a Producer**

****

There is a parameter acks, which is a guarantee of delivery. It has 3 possible values:

**acks = 0** – Fire and forget. producer doesn’t wait for a response.

**acks = 1** – Leader must receive the record and respond before write is considered successful.

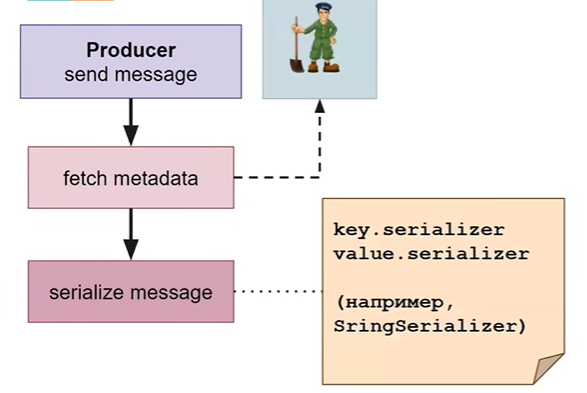
**acks = -1/all** – All online insync replicas must receive the write.

Based on broker and producer configuration, all three delivery semantics— “at most once”, “at least once” and “exactly once” — are supported.

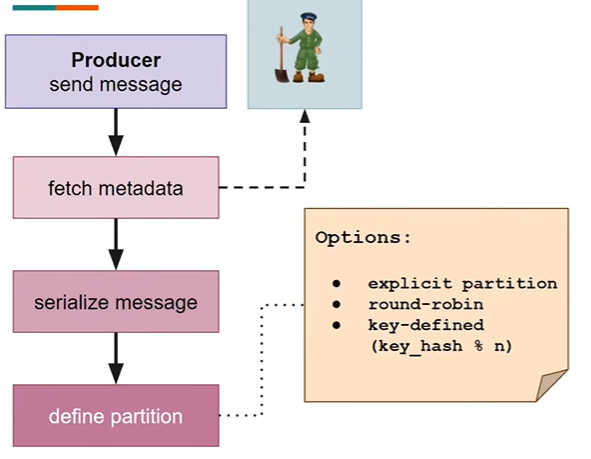
**At most once –** in this semantic, a message should be delivered maximum only once. It’s okay to loose a message rather then delivering it twice. You can use this semantic for log collection, metrics collection and so on. It has higher throughput and low latency.

**At least once –** In this semantic, message is delivered at least once. This is the most popular semantic and has moderate latency and throughput.

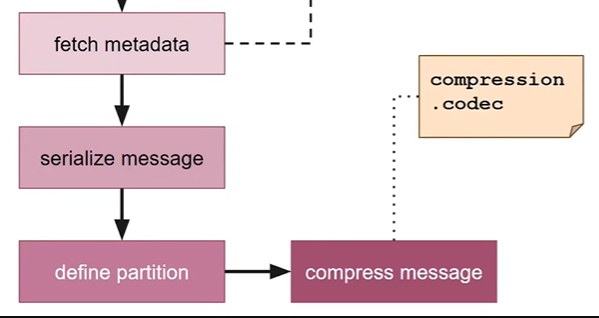
**Exactly once –** This semantic means that message should be delivered only once and no message should be lost. This is the most difficult one out of all, so has low throughput and high latency.



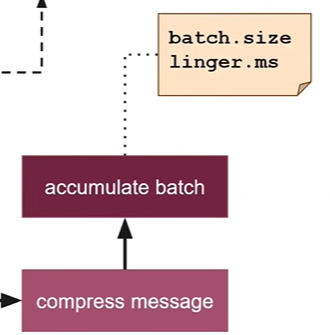
The next step is serializing the message. For example you can use StringSerializer.



Then you define a partition. You can explicitly use the partition number or not define it and kafka will use round-robin algorithm, or key base partition, when you use key and this key is hashed and module of n use used for defining on which partition it should be stored.



Then you compress The message. You can use different compression algorithms for it.



After that batch is accumulated. You configure batch size configuration and after this size is passed, batch is sent. Or you can configure linger.ms which waits for this time(ms) and sends it even if batch.size is not enough.

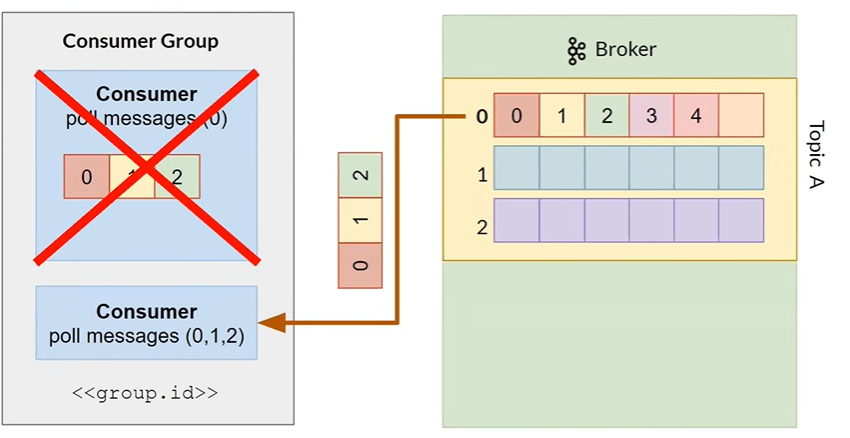
If on one broker, there are two batches created for different partitions, if the sum of the size of this batches are more then configured batch.size, it won’t wait and send this batch.

**Consumer**

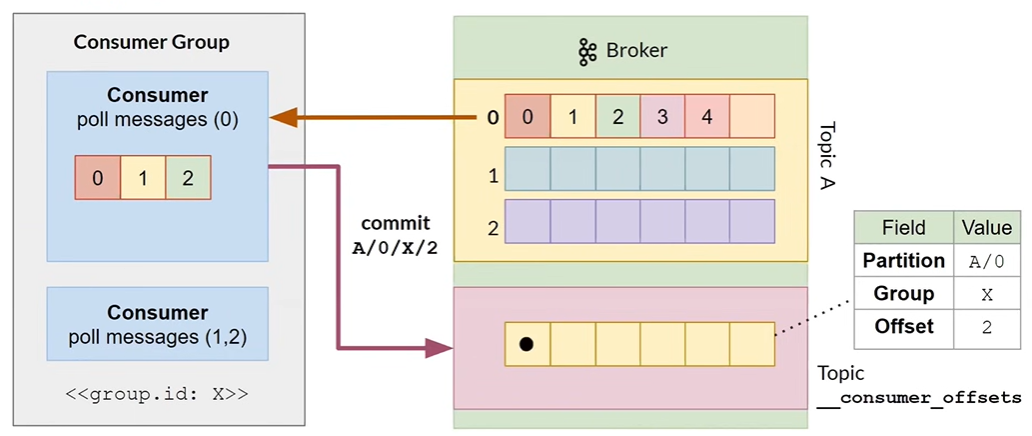
Consumer only reads from the leader partition. And it gets a package of messages rather then only one message.

You can group the consumers into Consumer Groups and distribute poll of partitions to this consumers. If you don’t do this one consumer will get messages of every partition.

There is one problem with Consumer Group. If a consumer fetched the data and it got down, another consumer will get the same messages to process, which creates a problem.



For fixing this, there is a topic for storing consumer offsets, called \_consumer\_offsets.



This topic stores partition, group and offset. And starts next commit from the latest offset + 1.

Types of commits:

**Auto commit:** at most once(miss messages)

**Manual commit:** at least once(duplicate messages)

**Custom offset management:** exactly once