A Kafka cluster is made of one or more servers. In the Kafka universe, they are called Brokers. Each broker holds a subset of records that belongs to the entire cluster.

**Kafka Brokers**

* A Kafka cluster is made up of multiple Kafka Brokers.
* Each Kafka Broker has a unique ID (number).
* Kafka Brokers contain topic log partitions. Connecting to one broker bootstraps a client to the entire Kafka cluster.
* For fail-over, you want to start with at least three brokers.

**Kafka needs ZooKeeper**

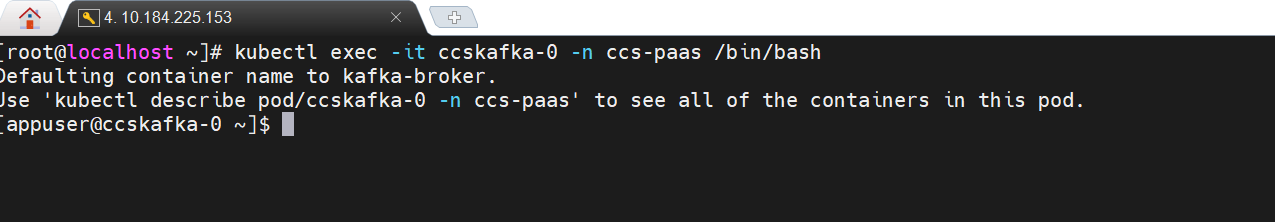
* Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs.
* Kafka uses Zookeeper to manage service discovery for Kafka Brokers that form the cluster.
* Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc.
* Zookeeper provides an in-sync view of Kafka Cluster configuration.

**Kafka Topic details**

* Each partition has one broker which acts as the "leader" and zero or more broker which act as "followers".
* The leader handles all read and write requests for the partition while the followers passively replicate the leader.

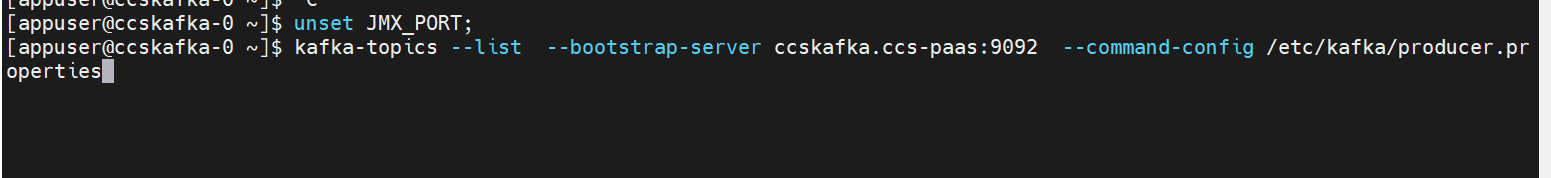
**List a Topic:**

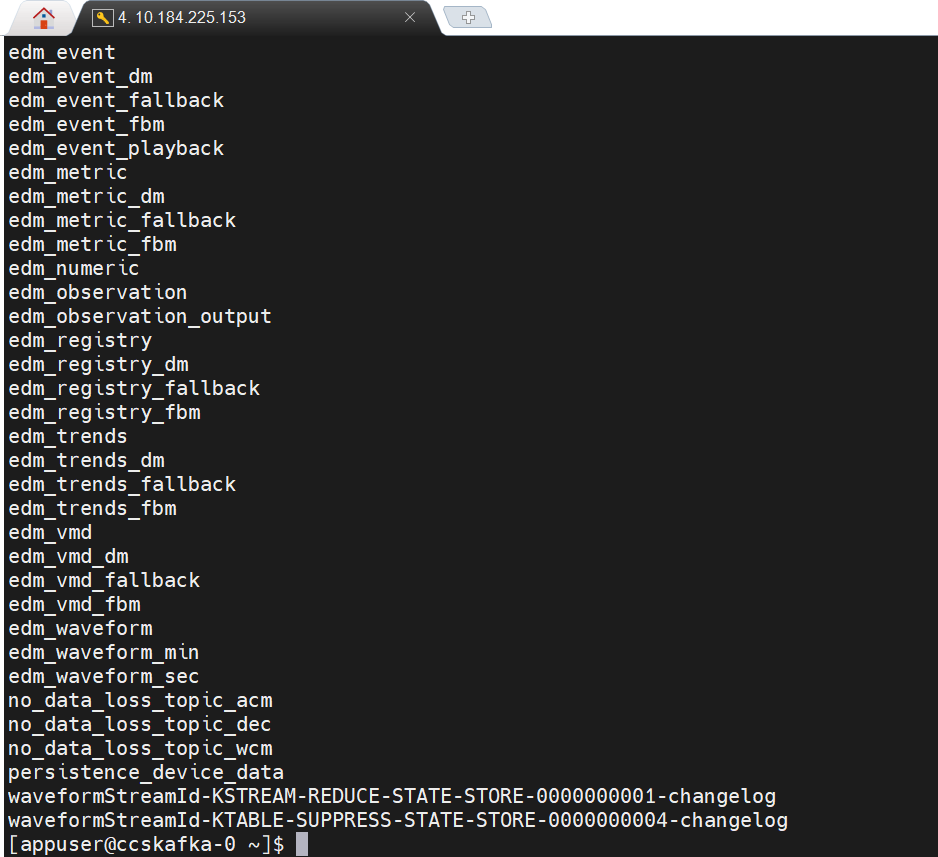
*kubectl exec -it ccskafka-0 -n ccs-paas /bin/bash*



*unset JMX\_PORT;*

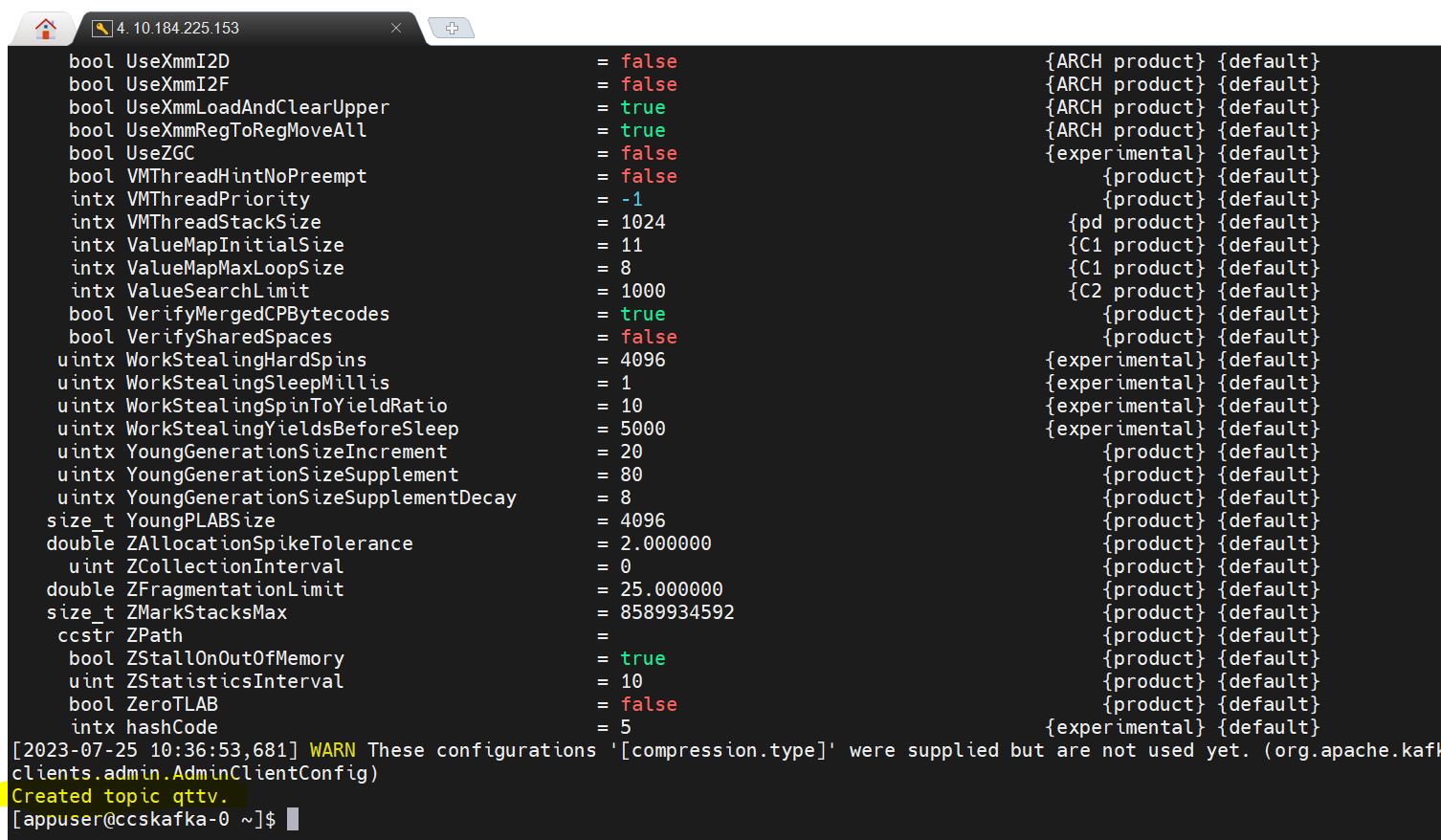
*kafka-topics --list --bootstrap-server ccskafka.ccs-paas:9092 --command-config /etc/kafka/producer.properties*





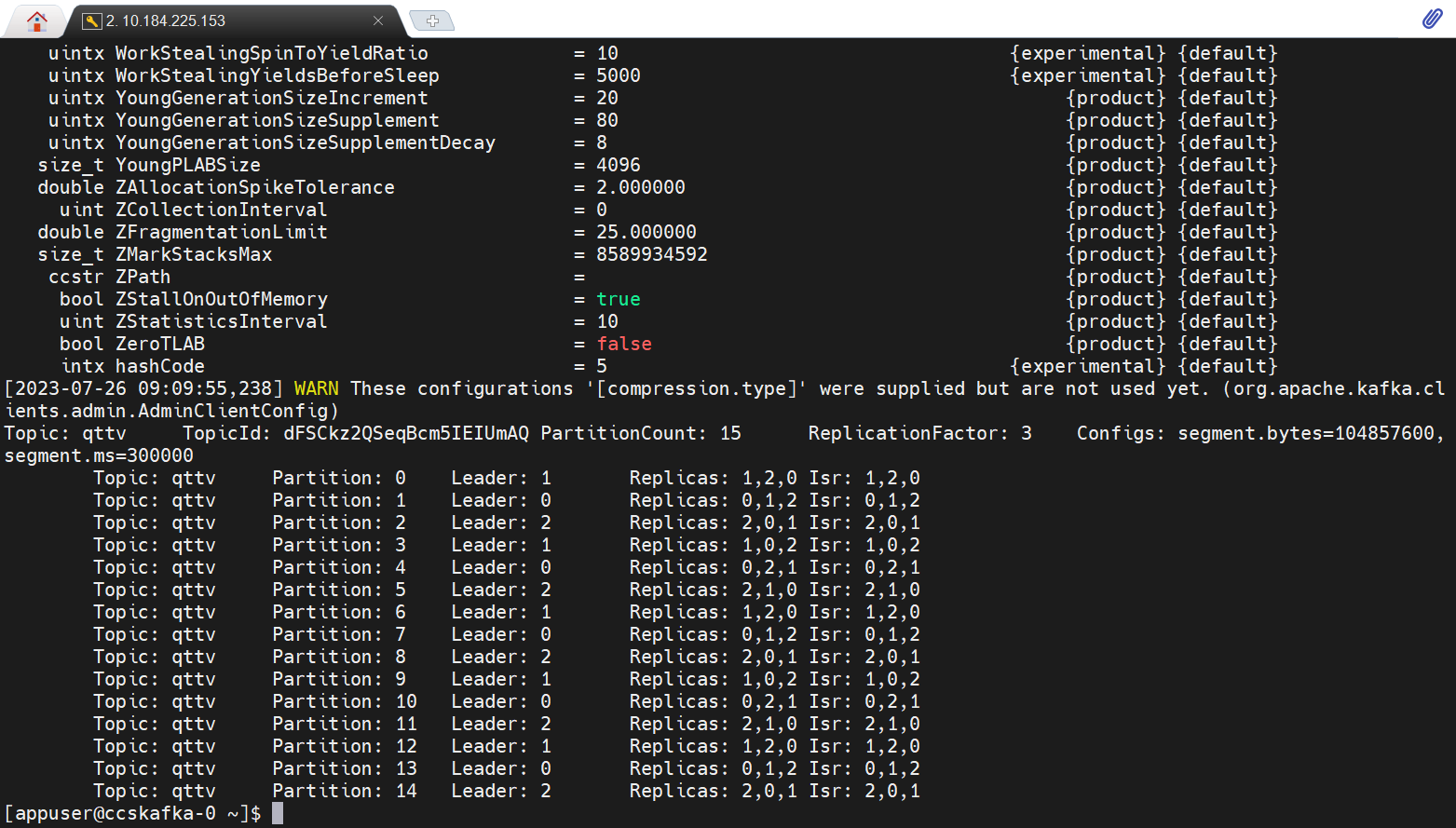
**Create topic:**

kafka-topics --create --bootstrap-server ccskafka.ccs-paas:9092 --replication-factor 3 --partitions 15 --topic qttv --command-config /etc/kafka/producer.properties



Desc Topic :

kafka-topics --describe --topic qttv --bootstrap-server ccskafka.ccs-paas:9092 --command-config /etc/kafka/producer.properties



**Delete topic:**

kafka-topics --delete --topic qttv --bootstrap-server ccskafka.ccs-paas:9092 --command-config /etc/kafka/producer.properties

**Send data or produce data to a topic :**

kafka-console-producer --topic qttv --broker-list ccskafka.ccs-paas:9092 --producer.config /etc/kafka/producer.properties

**Consume data from a topic:**

kafka-console-consumer --topic test --from-beginning --bootstrap-server ccskafka.ccs-paas:9092 --consumer.config /etc/kafka/consumer.properties

## **Topics**

When an event stream enters Kafka, it is persisted as a topic. In Kafka’s universe, a topic is a materialized event stream. In other words, a topic is a **stream at rest**.

Topic groups related events together and durably stores them. The closest analogy for a Kafka topic is a table in a database or folder in a file system.

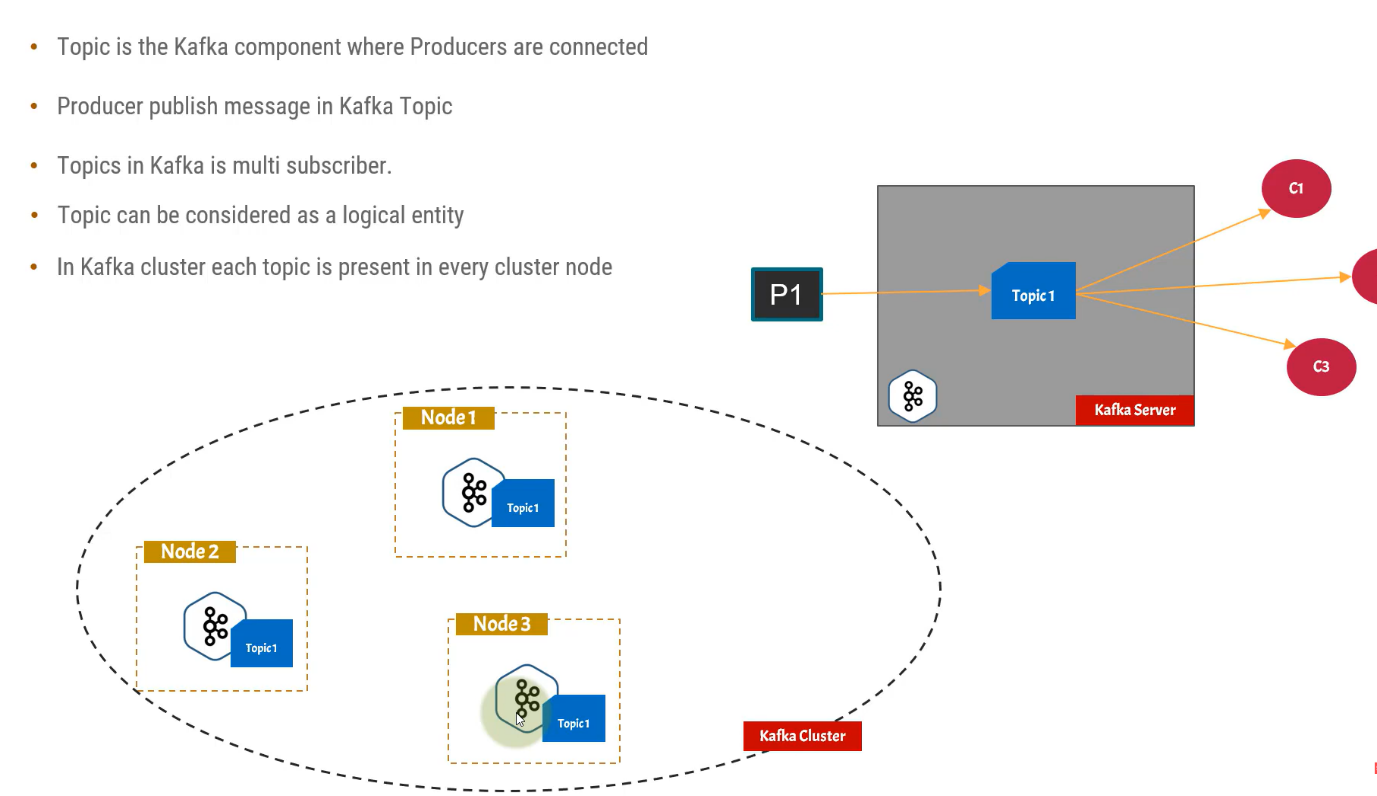
Topics are the central concept in Kafka that decouples producers and consumers. A consumer pulls messages off of a Kafka topic while producers push messages into a Kafka topic. A topic can have many producers and many consumers.

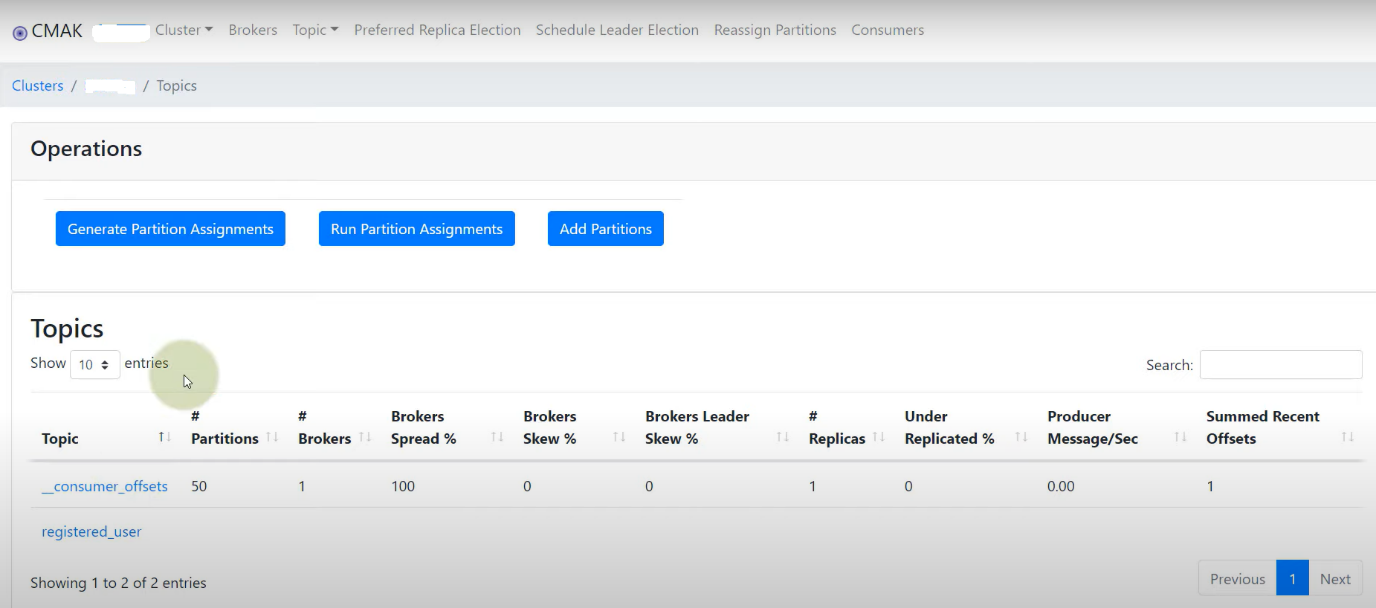
If a message includes a message key (such as a user ID or order ID), it will always be written to the same partition. A message key can be a string, number, object, or anything else, but it is optional; if no key is provided, the key will be set to null and the message will be sent in a round-robin fashion to different partitions. Kafka uses a key-hashing mechanism to distribute messages with keys to the appropriate partitions.

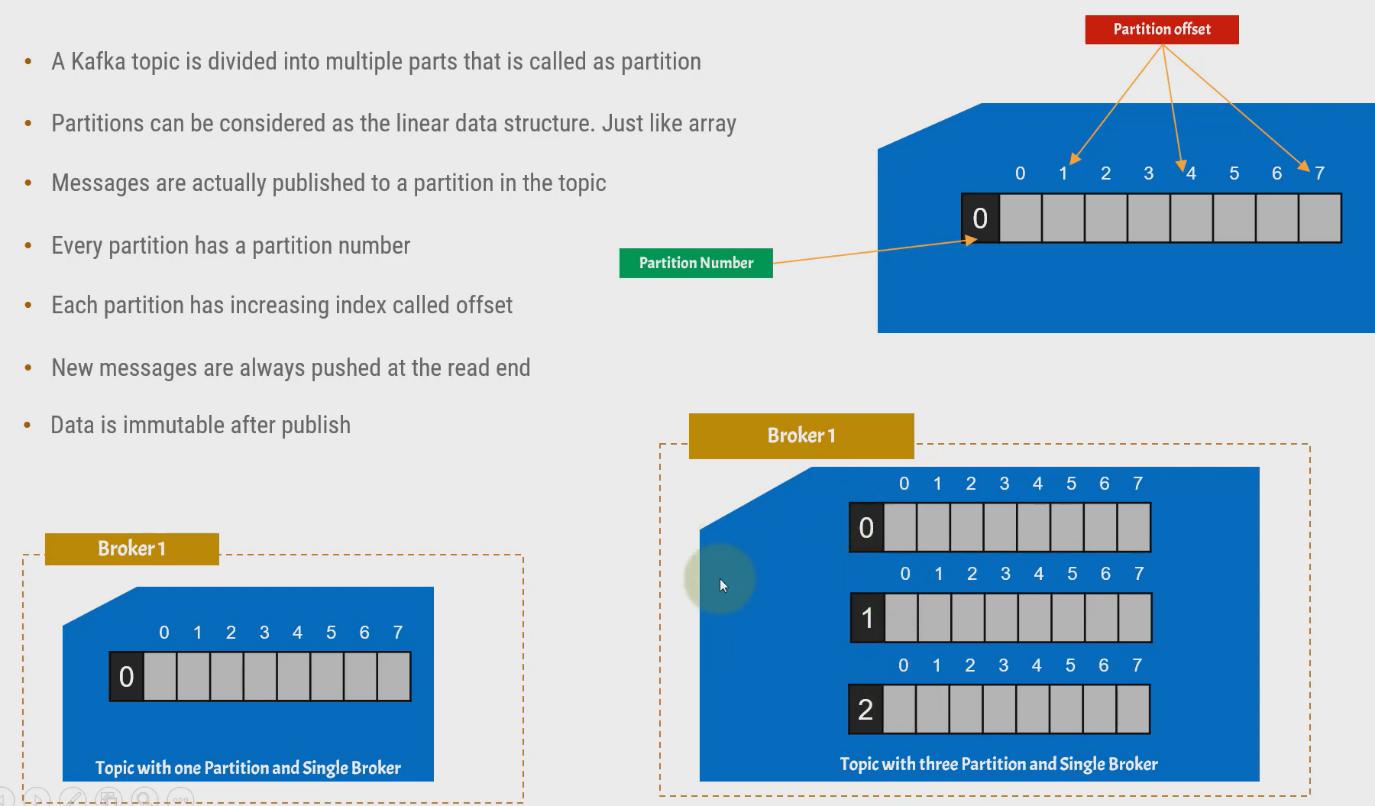
Once a message is written to a Kafka topic, it cannot be changed. This is called immutability, and it means that Kafka topics are immutable.

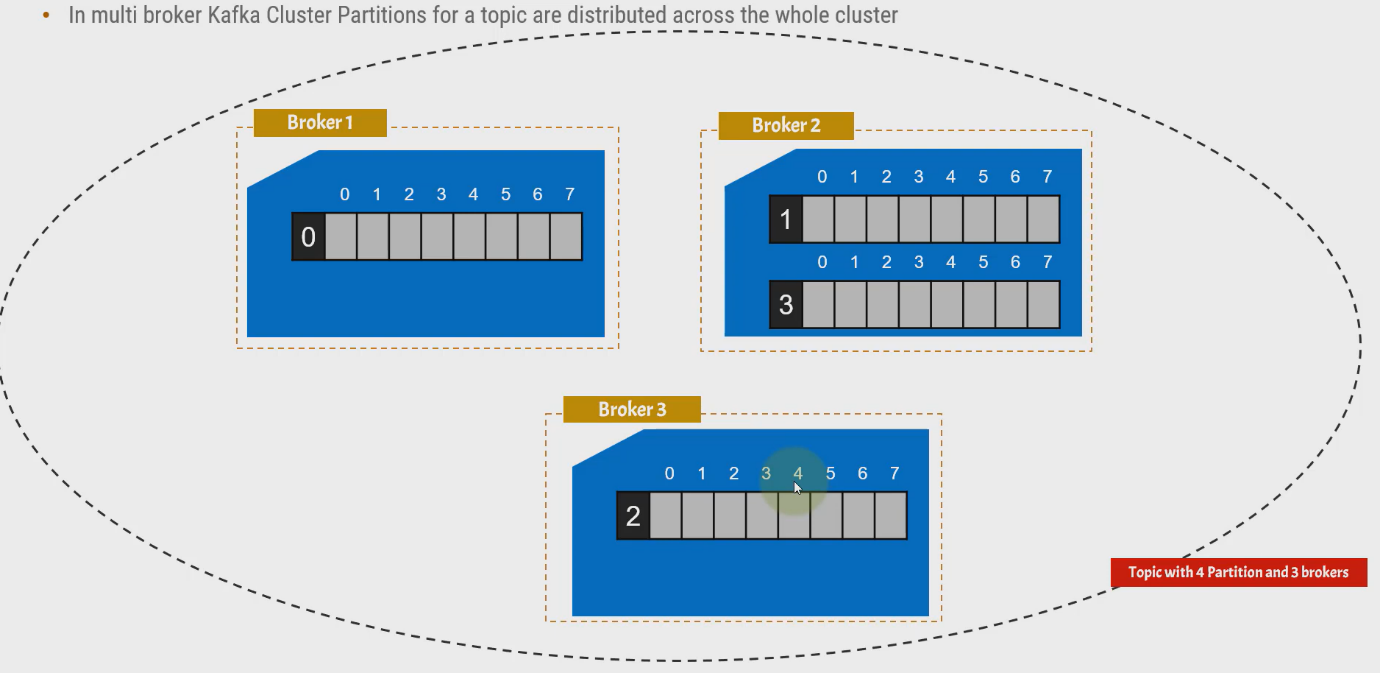
## **Replication factor:** While creating a topic use rep factor , always it must be equal to available broker or less than broker

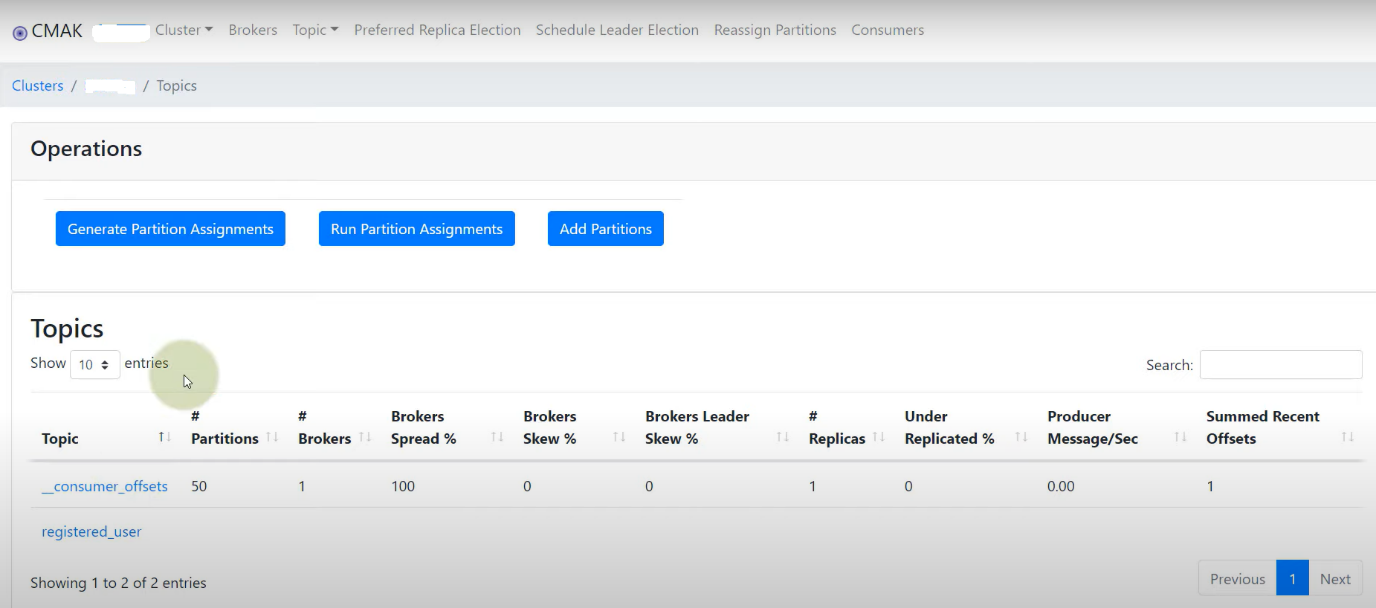
I got an error that says **Replication factor: 4 larger than available brokers: 3**.

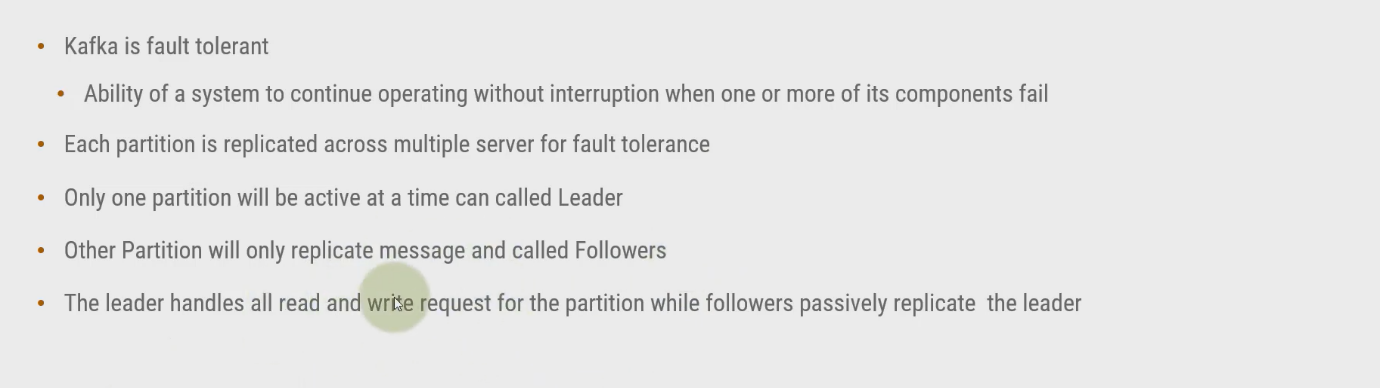


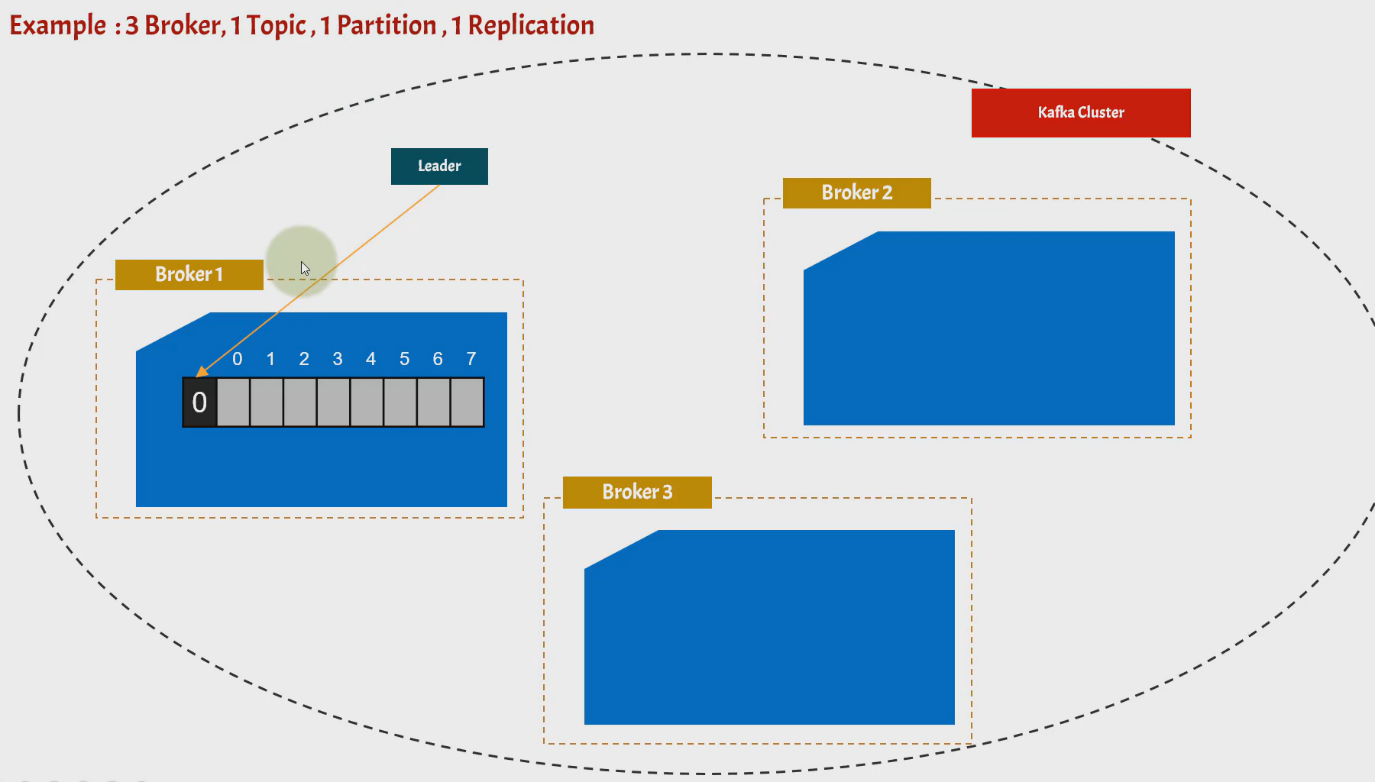


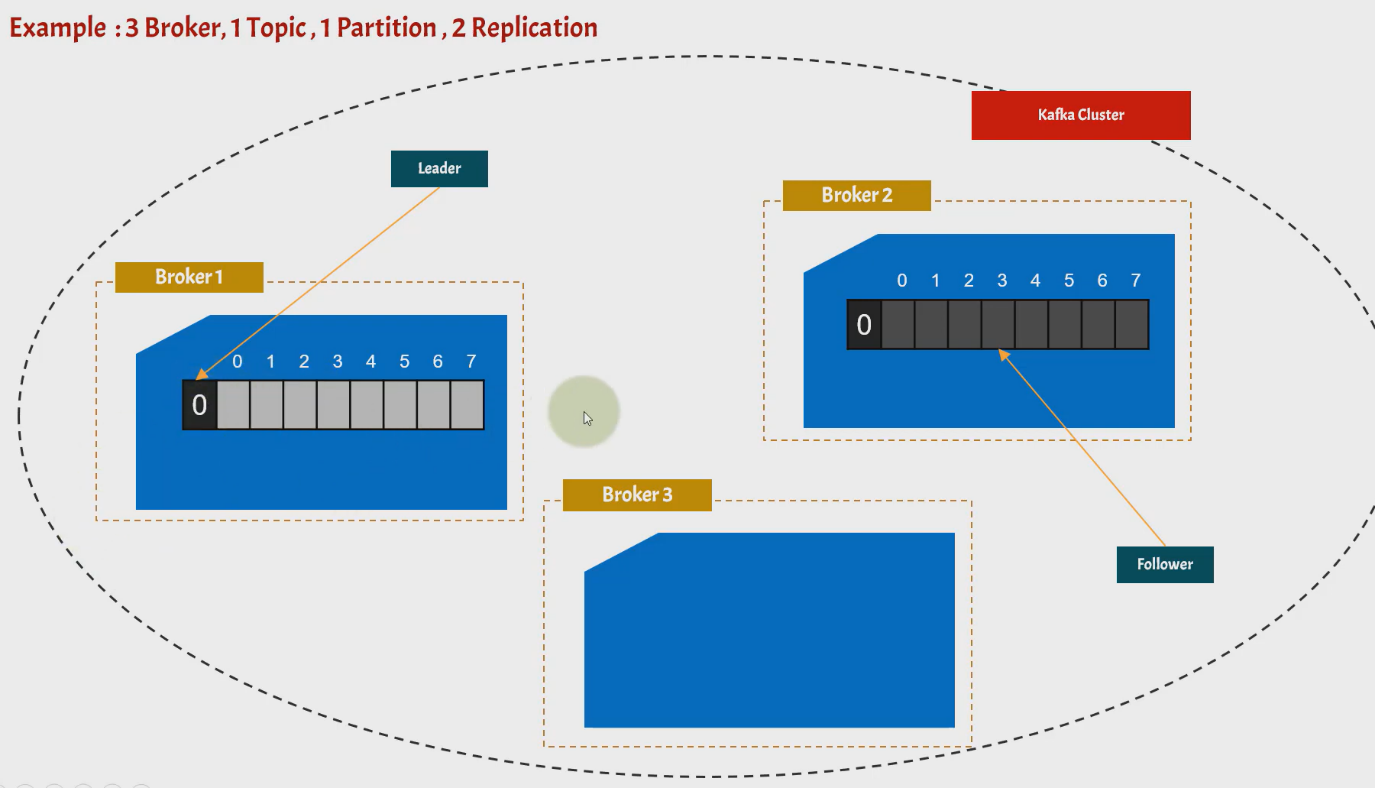


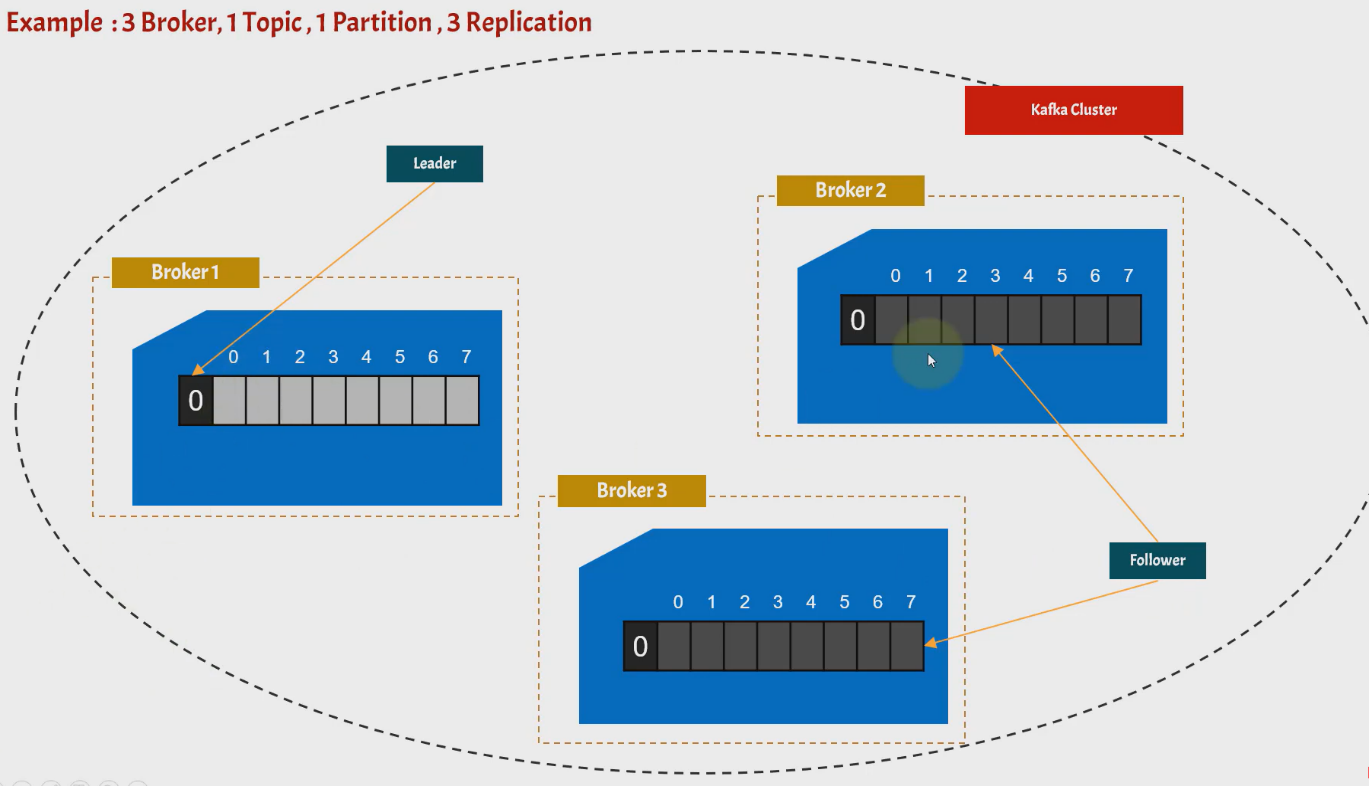


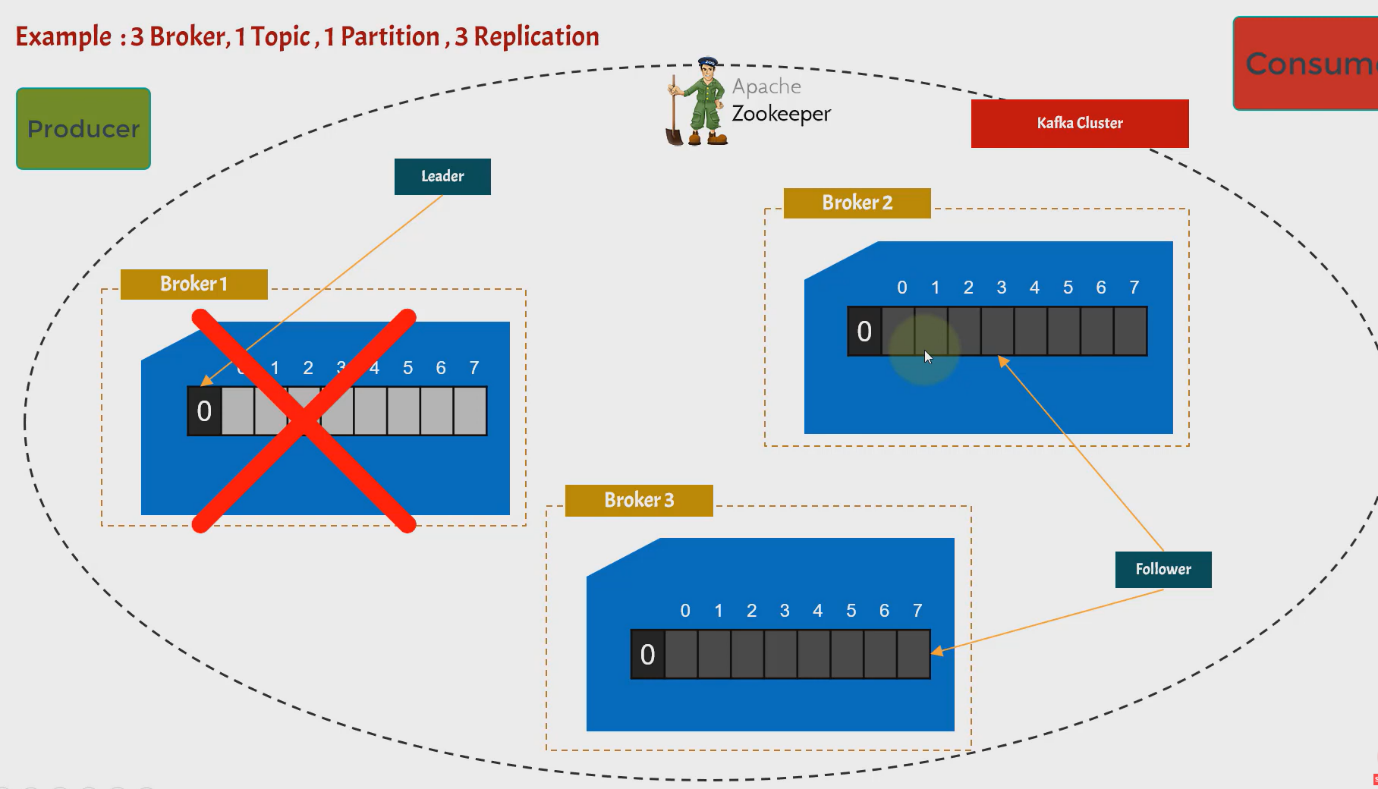












## **Offsets**

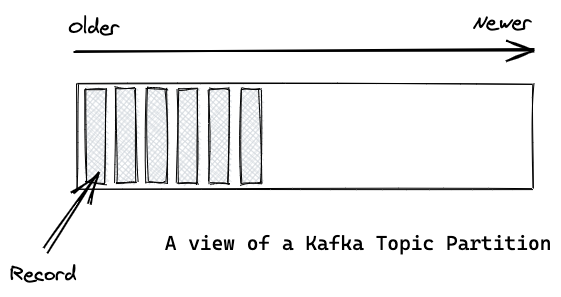
In Kafka, each message within a partition is assigned a unique sequential integer called an “offset.” The offset is used to identify each message within the partition. By default, the offset starts at 0 for each partition and is incremented by 1 each time a new message is received. However, offsets are specific to a partition, so the message with offset 1 in partition 0 may not be the same as the message with offset 1 in partition 1.

Kafka guarantees the order of messages within a partition, but there is no guarantee of order between messages in different partitions of the same topic. Additionally, Kafka messages are deleted over time, so once a message is deleted, its offset will no longer be used. Offsets in Kafka are continuously increasing in a never-ending sequence.

Kafka is designed to be fault-tolerant and highly resilient through data replication. In Kafka, topics can be replicated across multiple brokers, meaning that duplicate copies of the data are written to multiple servers. This prevents data loss in case of failure and ensures that the data is always available for consumption. Replication in Kafka is performed at the partition level, and the replication factor (i.e., the number of copies of the data) can be set at the time of topic creation. A replication factor of 3 is commonly used in production environments, which means that there will always be three copies of the data. A replication factor of 1 means that there is no replication

# **Partitions**

Kafka’s topics are divided into several **partitions**. While the topic is a logical concept in Kafka, a partition is the smallest storage unit that **holds a subset of records owned by a topic**. Each partition is a single log file where records are written to it in an append-only fashion.



When talking about the content inside a partition, I will use the terms **record** and **message** interchangeably.

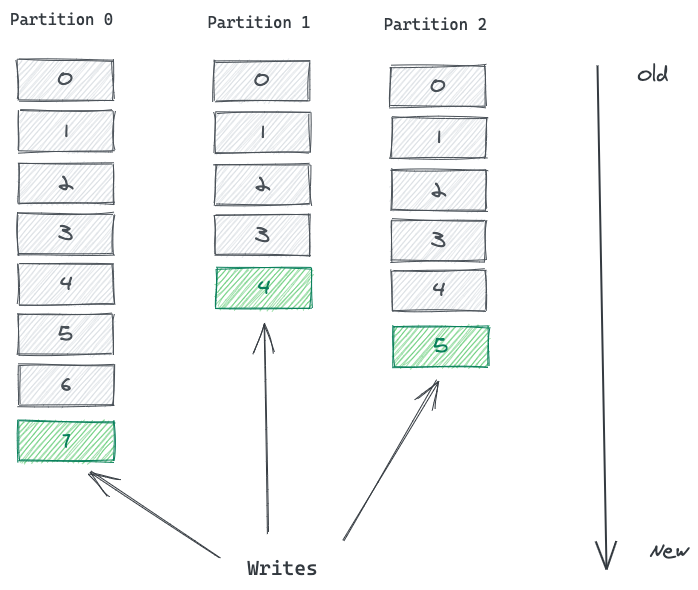
# Offsets and the ordering of messages

The records in the partitions are each assigned a sequential identifier called the offset, which is unique for each record within the partition.

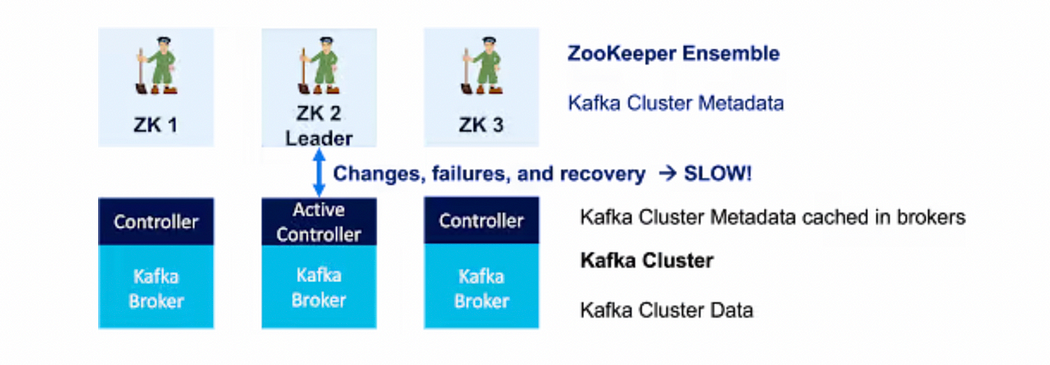
The offset is an incremental and immutable number, maintained by Kafka. When a record is written to a partition, it is appended to the end of the log, assigning the next sequential offset. Offsets are particularly useful for consumers when reading records from a partition. We’ll come to that at a later point.

The figure below shows a topic with three partitions. Records are being appended to the end of each one.

Although the messages within a partition are ordered, messages across a topic are not guaranteed to be ordered.



A topic in Kafka is broken into multiple partitions



**Leader Election:**

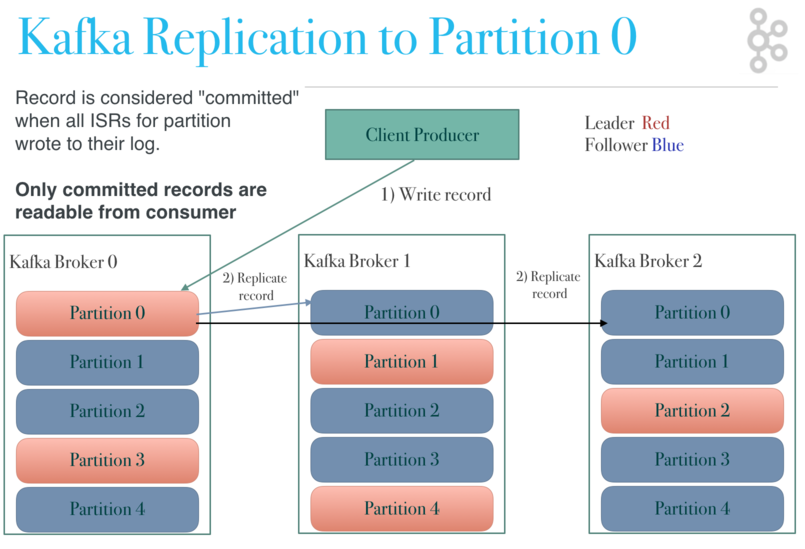
leader election is the process of selecting a new leader for a partition when the current leader fails or becomes unavailable

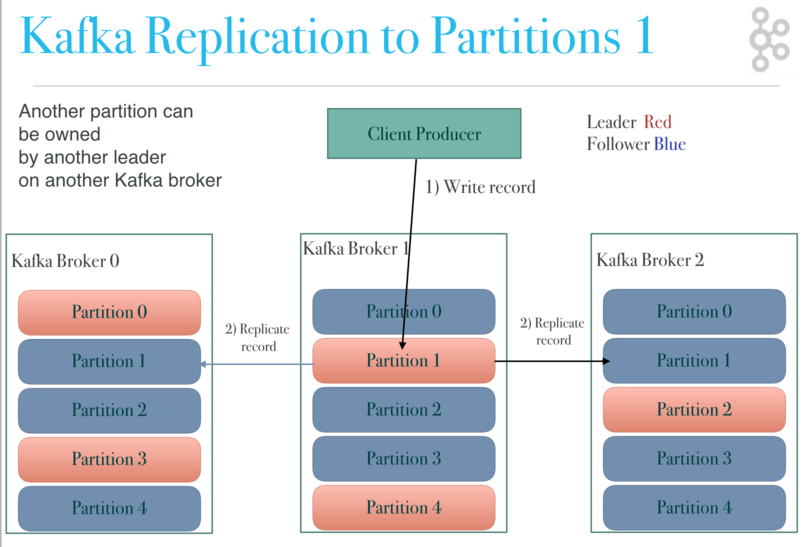
Kafka’s **fault-tolerance** (ensures that data is not lost and the cluster continues to operate smoothly.)

1. Kafka uses Quorum based **Consensus algorithm** to ensure that only one broker becomes the leader for a given partition.

A quorum is a minimum number of votes required to make a decision.

In Kafka, the quorum **size** is determined by the **replication** **factor** of the partition



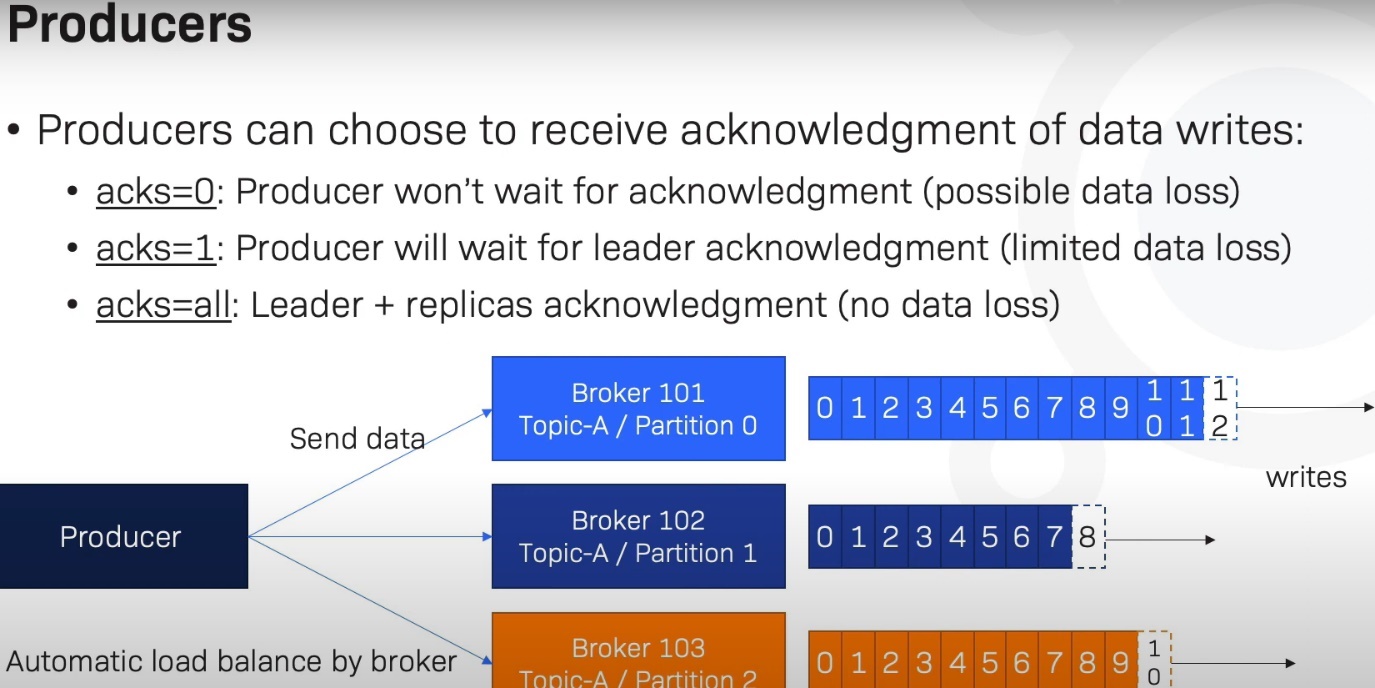


#### failover for consumers

If a consumer in a consumer group dies, the partitions assigned to that consumer is divided up amongst the remaining consumers in that group.

#### failover for Brokers

If a broker dies, then Kafka divides up leadership of its topic partitions to the remaining brokers in the cluster.

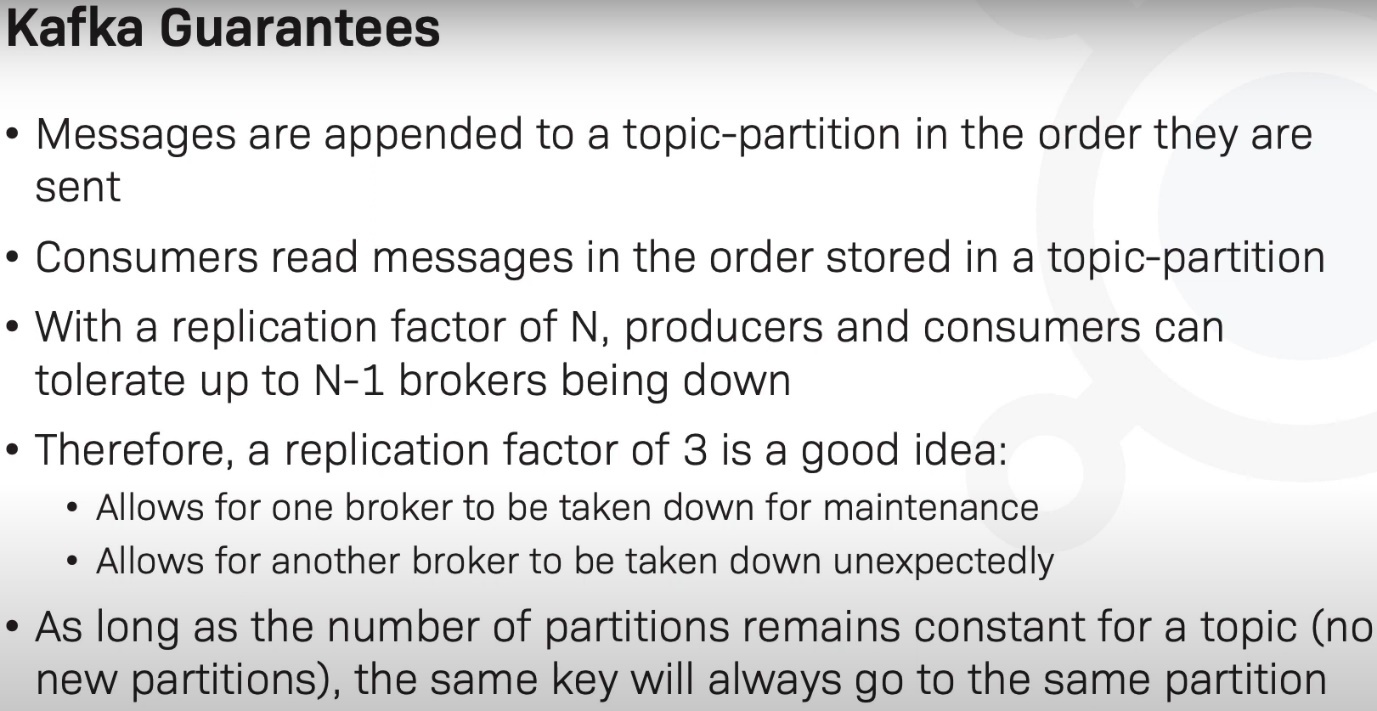


**Recommended Producer configuration:**

Exactly once in order semantics per partition

An idempotent operation is one which can be performed many times without causing a different effect than only being performed once. The producer send operation is now idempotent. In the event of an error that causes a producer retry, the same message—which is still sent by the producer multiple times—will only be written to the Kafka log on the broker once.

To turn on this feature and get exactly once semantics per partition—meaning no duplicates, no data loss, and in-order semantics—configure your producer to set “enable.idempotence=true”



Check Zookeeper leader and followers

echo stat | nc localhost:2181

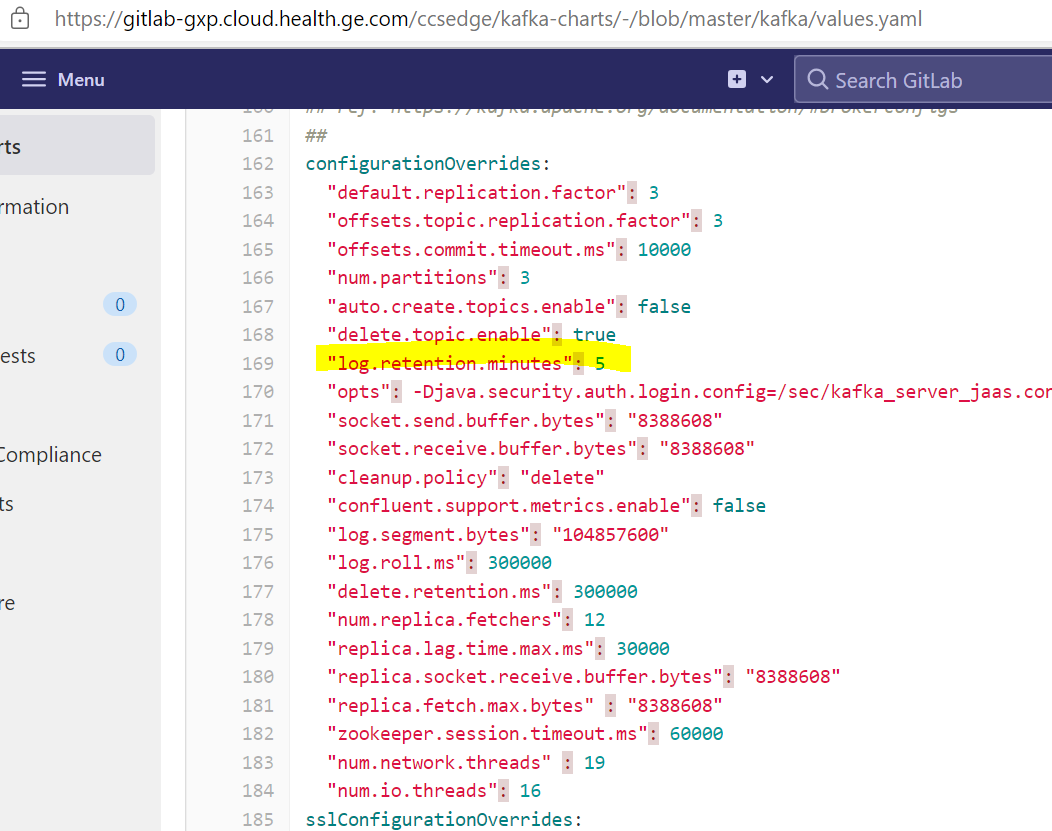
echo stat | nc localhost:2182

echo stat | nc localhost:2183

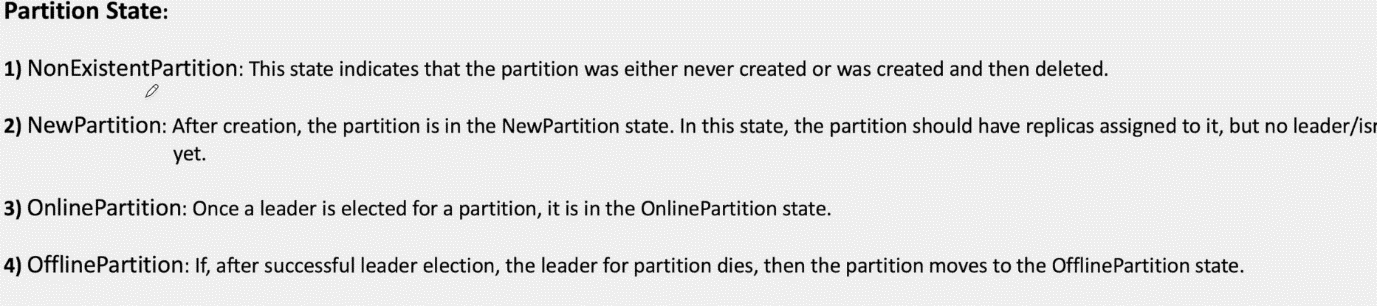
check brokers are connected with Zookeeper or not :

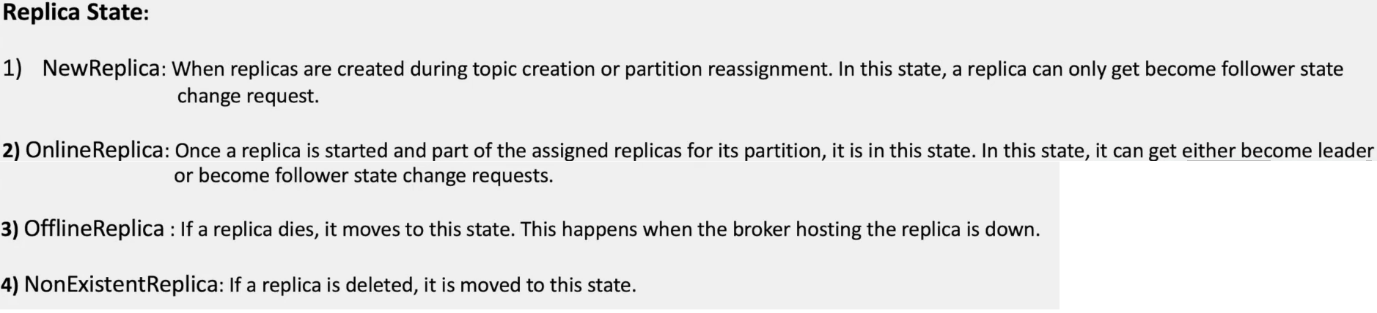
echo dump | nc localhost 2181

<https://gitlab-gxp.cloud.health.ge.com/ccsedge/kafka-charts/-/blob/master/kafka/values.yaml>

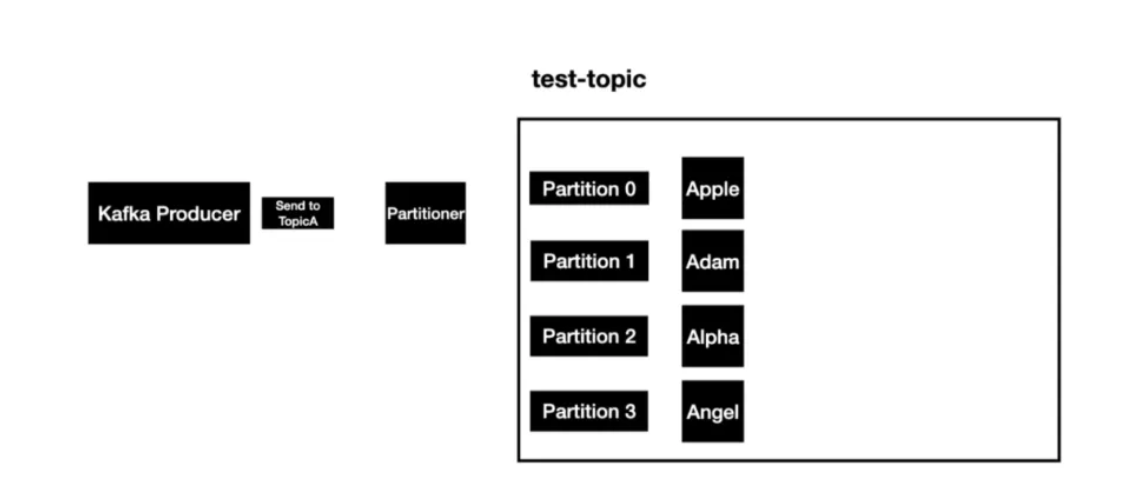


**Controller manages all internal States of partition and Replicas**



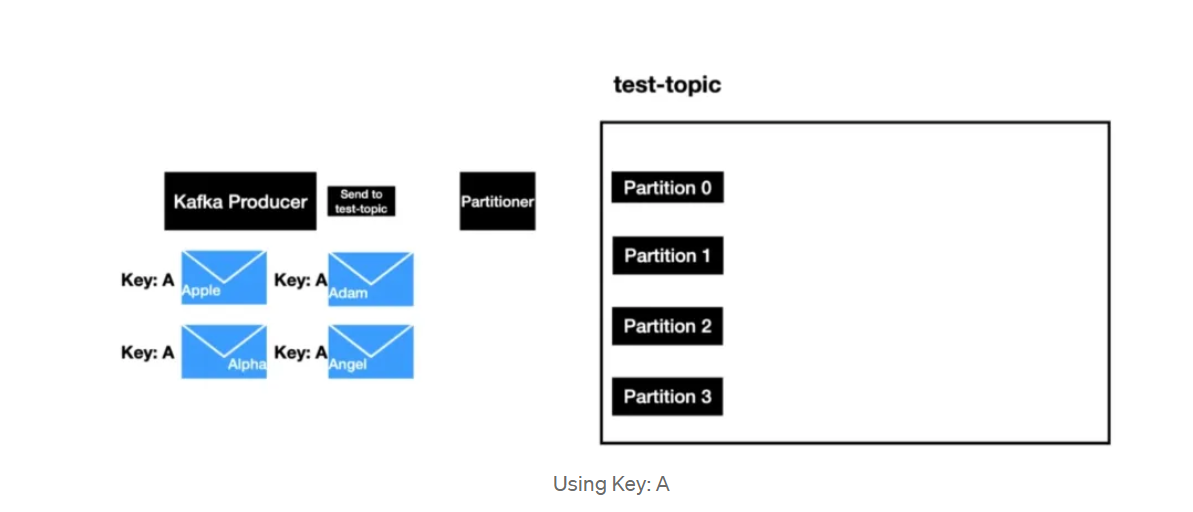


**Without key** (the partitioner will use the Round Robin algorithm to send the messages)



**With key** -When a message is sent, Kafka Partioner is going to apply some hashing techniques to determine the partition value and if the same key is sent then it is going to resolve to the same partitions. **Partition-0** in these cases.

Same keys



Different key :

