# What is Apache Kafka?

Apache Kafka is a software platform which is based on a distributed streaming process.

It is a publish-subscribe messaging system which lets exchanging of data between applications, servers, and processors as well.

Apache Kafka was originally developed by **LinkedIn**, and later it was donated to the Apache Software Foundation.

Currently, it is maintained by **Confluent** under Apache Software Foundation.

Apache Kafka has resolved the lethargic trouble of data communication between a sender and a receiver.

# 

# What is a messaging system?

A messaging system is a simple exchange of messages between two or more persons, devices, etc.

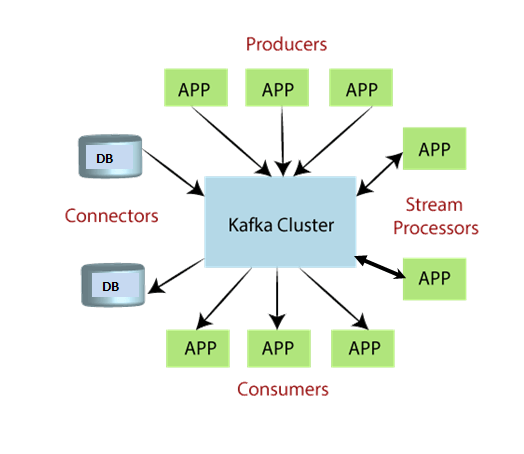
A publish-subscribe messaging system allows a sender to send/write the message and a receiver to read that message.

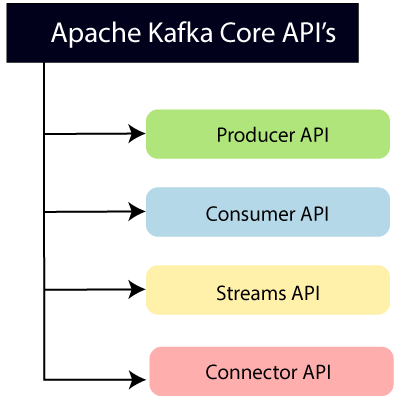
In Apache Kafka, a sender is known as a **producer** who publishes messages, and a receiver is known as a **consumer** who consumes that message by subscribing it.

# What is a streaming process?

A streaming process is the processing of data in parallelly connected systems.

This process allows different applications to limit the parallel execution of the data, where one record executes without waiting for the output of the previous record.





**Producer API:** This API allows/permits an application to publish streams of records to one or more topics. (discussed in later section)

**Consumer API:** This API allows an application to subscribe to one or more topics and process the stream of records produced to them.

**Streams API:** This API allows an application to effectively transform the input streams to the output streams. It permits an application to act as a stream processor which consumes an input stream from one or more topics, and produces an output stream to one or more output topics.

**Connector API:** This API executes the reusable producer and consumer APIs with the existing data systems or applications.

# 

# Kafka Words

## Topics:

In Kafka, the word topic refers to a category or a common name used to store and publish a particular stream of data.

Basically, topics in Kafka are similar to tables in the database, but not containing all constraints.

In Kafka, we can create as many topics as we want.

It is identified by its name, which depends on the user's choice.

A producer publishes data to the topics, and a consumer reads that data from the topic by subscribing it.

## Partitions

A topic is split into several parts which are known as the partitions of the topic.

These partitions are separated in an order.

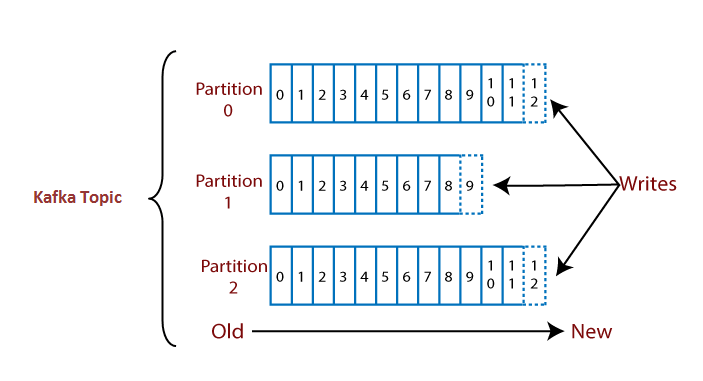
The data content gets stored in the partitions within the topic.

Therefore, while creating a topic, we need to specify the number of partitions(the number is arbitrary and can be changed later).

Each message gets stored into partitions with an incremental id known as its Offset value.

The order of the **offset value** is guaranteed within the partition only and not across the partition.

The offsets for a partition are infinite.



Suppose, a topic containing three partitions 0,1 and 2.

Each partition has different offset numbers.

The data is distributed among each offset in each partition where data in offset 1 of Partition 0 does not have any relation with the data in offset 1 of Partition1.

But, data in offset 1of Partition 0 is interrelated with the data contained in offset 2 of Partition0.

## Brokers

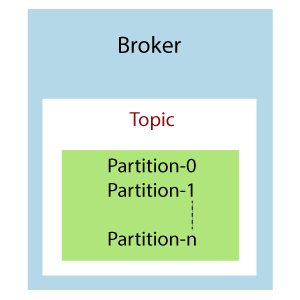
A Kafka cluster consists of one or more servers which are known as brokers or Kafka brokers.

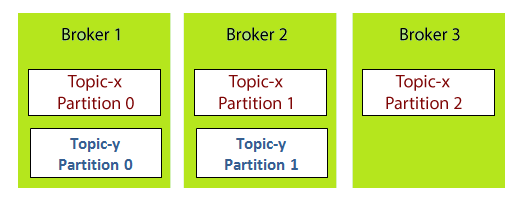
A broker is a container that holds several topics with their multiple partitions.

The brokers in the cluster are identified by an integer id only.

Kafka brokers are also known as **Bootstrap brokers** because connection with any one broker means connection with the entire cluster.

Although a broker does not contain whole data, each broker in the cluster knows about all other brokers, partitions as well as topics.





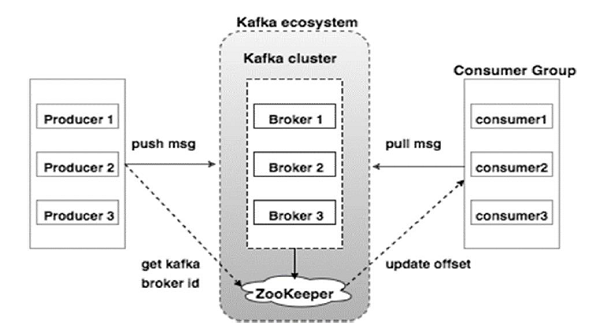
Each broker is holding a topic, namely Topic-x with three partitions 0,1 and 2.

Remember, all partitions do not belong to one broker only, it is always distributed among each broker (depends on the quantity).

Broker 1 and Broker 2 contains another topic-y having two partitions 0 and 1.

Thus, Broker 3 does not hold any data from Topic-y.

It is also concluded that no relationship ever exists between the broker number and the partition number.



# Kafka Replication

Apache Kafka is a distributed software system in the Big Data world.

Thus, for such a system, there is a requirement to have copies of the stored data.

In Kafka, each broker contains some sort of data.

But, what if the broker or the machine fails down? The data will be lost.

Precautionary, Apache Kafka enables a feature of replication to secure data loss even when a broker fails down.

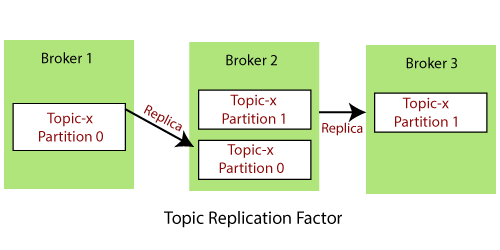
To do so, a **replication factor** is created for the topics contained in any particular broker.

A replication factor is the number of copies of data over multiple brokers.

The replication factor value should be greater than 1 always (between 2 or 3).

This helps to store a replica of the data in another broker from where the user can access it.

For example, suppose we have a cluster containing three brokers say Broker 1, Broker 2, and Broker 3. A topic, namely Topic-X is split into Partition 0 and Partition 1 with a replication factor of 2.



Thus, we can see that Partition 0 of Topic-x is having its replicas in Broker 1 and Broker 2. Also, Partition1 of Topic-x is having its replication in Broker 2 and Broker 3.

It is obvious to have confusion when both the actual data and its replicas are present. The cluster may get confused about which broker should serve the client request. To remove such confusion, the following task is done by Kafka:

1. It chooses one of the broker's partitions as a leader, and the rest of them becomes its followers.
2. The followers(brokers) will be allowed to synchronize the data. But, in the presence of a **leader**, none of the **followers** is allowed to serve the client's request. These replicas are known as ISR(in-sync-replica). So, Apache Kafka offers multiple ISR(in-sync-replica) for the data.

Therefore, only the leader is allowed to serve the client request.

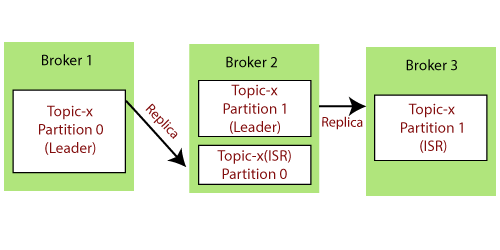
The leader handles all the read and writes operations of data for the partitions.

The leader and its followers are determined by the zookeeper(discussed later).

If the broker holding the leader for the partition fails to serve the data due to any failure, one of its respective ISR replicas will take over the leadership. Afterward, if the previous leader returns back, it tries to acquire its leadership again.

Let's see an example to understand the concept of a leader and its followers.

Suppose, a cluster with the following three brokers 1,2, and 3. A topic x is present having two partitions and with replication factor=2.



So, to remove the confusion, Partition-0 under Broker 1 is provided with the leadership.

Thus, it is the leader and Partition 0 under Broker 2 will become its replica or ISR. Similarly,

Partition 1 under Broker 2 is the leader and Partition 1 under Broker 3 is its replica or ISR.

In case, Broker 1 fails to serve, Broker 2 with Partition 0 replica will become the leader.

# Kafka Producer

A producer is the one which publishes or writes data to the topics within different partitions.

Producers automatically know what data should be written to which partition and broker.

The user does not require to specify the broker and the partition.

## How does the producer write data to the cluster?

A producer uses following strategies to write data to the cluster:

* Message Keys
* Acknowledgment

### Message Keys

Apache Kafka enables the concept of the key to send the messages in a specific order.

The key enables the producer with two choices, i.e., either to send data to each partition (automatically) or send data to a specific partition only.

Sending data to some specific partitions is possible with the message keys.

If the producers apply a key over the data, that data will always be sent to the same partition always.

But, if the producer does not apply the key while writing the data, it will be sent in a round-robin manner. This process is called **load balancing**.

In Kafka, load balancing is done when the producer writes data to the Kafka topic without specifying any key, Kafka distributes little-little bit data to each partition.

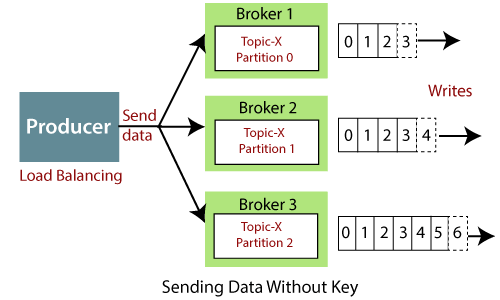
Therefore, a message key can be a string, number, or anything as we wish.

There are two ways to know that the data is sent with or without a key:

1. If the value of key=NULL, it means that the data is sent without a key. Thus, it will be distributed in a round-robin manner (i.e., distributed to each partition).
2. If the value of the key!=NULL, it means the key is attached with the data, and thus all messages will always be delivered to the same partition.

Consider a scenario where a producer writes data to the Kafka cluster, and the data is written without specifying the key.

So, the data gets distributed among each partition of Topic-T under each broker, i.e., Broker 1, Broker2, and Broker 3.

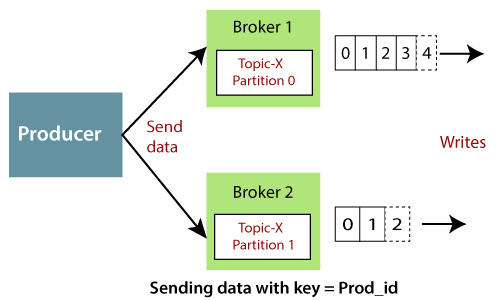


Consider another scenario where a producer specifies a key as Prod\_id.

So, data of Prod\_id\_1(say) will always be sent to partition 0 under Broker 1, and data of

Prod\_id\_2 will always be in partition 1 under Broker 2.

Thus, the data will not be distributed to each partition after applying the key (as saw in the above scenario).



### Acknowledgment

In order to write data to the Kafka cluster, the producer has another choice of acknowledgment.

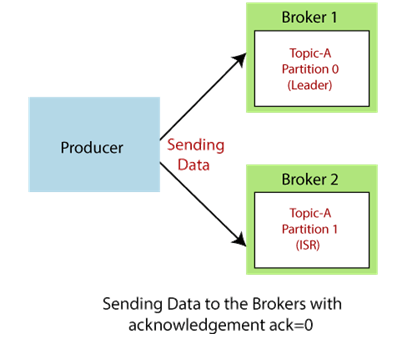
It means the producer can get a confirmation of its data writes by receiving the following acknowledgments:

* **acks=0:** This means that the producer sends the data to the broker but does not wait for the acknowledgement. This leads to possible data loss because without confirming that the data is successfully sent to the broker or maybe the broker is down, it sends another one.
* **acks=1:** This means that the producer will wait for the leader's acknowledgement. The leader asks the broker whether it successfully received the data, and then returns feedback to the producer. In such cases, there is limited data loss only.
* **acks=all:** Here, the acknowledgment is done by both the leader and its followers. When they successfully acknowledge the data, it means the data is successfully received. In this case, there is no data loss.

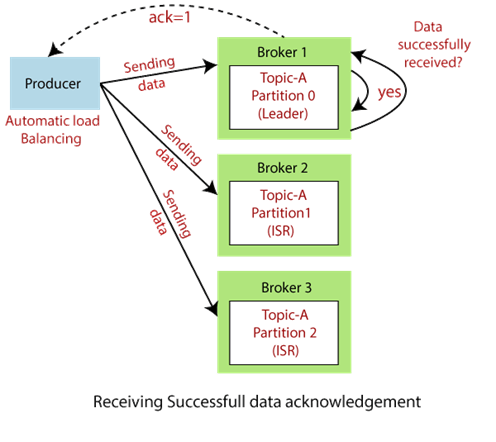
**Let' see an example**

Suppose, a producer writes data to Broker1, Broker 2, and Broker 3.

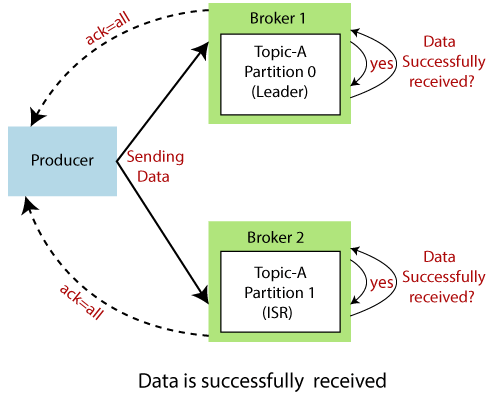
**Case1:** Producer sends data to each of the Broker, but not receiving any acknowledgement. Therefore, there can be a severe data loss, and the correct data could not be conveyed to the consumers.



**Case2:** The producers send data to the brokers. Broker 1 holds the leader. Thus, the leader asks Broker 1 whether it has successfully received data. After receiving the Broker's confirmation, the leader sends the feedback to the Producer with ack=1.



**Case3:** The producers send data to each broker. Now, the leader and its replica/ISR will ask their respective brokers about the data. Finally, acknowledge the producer with the feedback.



# Kafka: Consumer and Consumer Groups

A consumer is the one that consumes or reads data from the Kafka cluster via a topic.

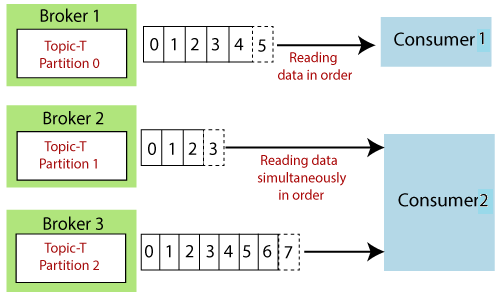
A consumer also knows that from which broker, it should read the data.

The consumer reads the data within each partition in an orderly manner.

It means that the consumer is not supposed to read data from offset 1 before reading from offset 0.

Also, a consumer can easily read data from multiple brokers at the same time

**For example,** two consumers namely, Consumer 1 and Consumer 2 are reading data. Consumer 1 is reading data from Broker 1 in sequential order. On the other hand, Consumer 2 is simultaneously reading data from Broker 2 as well as Broker 3 in order.



## Consumer Groups

A consumer group is a group of multiple consumers which are basically visions to an application basically.

Each consumer present in a group reads data directly from the exclusive partitions.

In case, the number of consumers is more than the number of partitions, some of the consumers will be in an inactive state.

Somehow, if we lose any active consumer within the group then the inactive one can take over and will come in an active state to read the data.

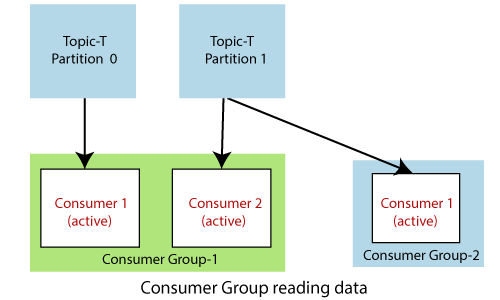
But, how to decide which consumer should read data first and from which partition?

For such decisions, consumers within a group automatically use a '**GroupCoordinator**' and one '**ConsumerCoordinator**', which assigns a consumer to a partition.

This feature is already implemented in Kafka. Therefore, the user does not need to worry about it.

**Example 1**

Consider two groups of consumers, i.e., Consumer Group-1 and Consumer Group-2. Both the consumers of Group 1 are reading data together but from different partitions. Both the consumers of Group 1 will remain in an active state because they are reading the data parallelly.



On the other hand, Consumer 1 of Group 2 is also reading the data from Partition 1 under Topic-T. Here, also the consumer is present in an active state because it belongs to Group 2.

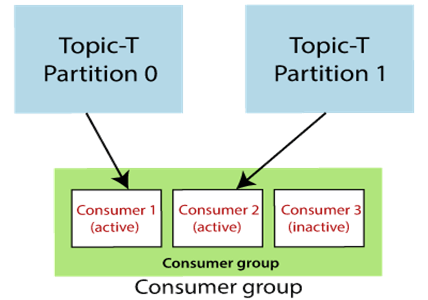
**Example 2**

Consider another scenario where a consumer group has three consumers.

Consumer 1 and Consumer 2 are present in an active state. Consumer 1 is reading data from Partition 0 and Consumer 2 from Partition 1.

As, there are only two topic-partitions available, but three consumers.

Thus, Consumer 3 will remain in an inactive state until any of the active consumers leaves.



## Consumer Offsets

Apache Kafka provides a convenient feature to store an offset value for a consumer group.

It stores an offset value to know at which partition, the consumer group is reading the data.

As soon as a consumer in a group reads data, Kafka automatically commits the offsets, or it can be programmed.

These offsets are committed live in a topic known as **\_\_consumer\_offsets**.

This feature was implemented in the case of a machine failure where a consumer fails to read the data.

So, the consumer will be able to continue reading from where it left off due to the commitment of the offset.

# **For example,**

# In the below figure, a consumer from a consumer group is reading the data. After reading the data, the consumer has committed the offset. It means next time, the consumer will read data not from the beginning but from the committed point. Also, somehow the consumer dies, it will be able to continue from the committed state only.

# Kafka Consumer and Consumer Groups

# 

# 

# Kafka installation

## Reference URL

<https://www.javatpoint.com/installation-of-apache-kafka>

# Creating Kafka Topics

<https://www.javatpoint.com/creating-kafka-topics>

Start zookeeper

C:\kafka>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

Start kafka server

C:\kafka>.\bin\windows\kafka-server-start.bat .\config\server.properties

Creating topic

C:\>kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic myfirst --create --partitions 3 --replication-factor 1

Checking topics list

kafka-topics.bat --zookeeper 127.0.0.1:2181 --list

Describing a topic

C:\>kafka-topics.bat --zookeeper 127.0.0.1:2181 --describe --topic myfirst

Topic: myfirst PartitionCount: 3 ReplicationFactor: 1 Configs:

Topic: myfirst Partition: 0 Leader: 0 Replicas: 0 Isr: 0

Topic: myfirst Partition: 1 Leader: 0 Replicas: 0 Isr: 0

Topic: myfirst Partition: 2 Leader: 0 Replicas: 0 Isr: 0

# Sending data to Kafka Topics

Without key

C:\>kafka-console-producer.bat --broker-list 127.0.0.1:9092 --topic myfirst

>Hello

>First Producer

>Ratna Created it

With key

C:\>kafka-console-producer.bat --broker-list 127.0.0.1:9092 --topic myfirst --p

roperty parse.key=true --property key.separator=,

>name,ratna

>age,27

>location,hyd

# Kafka Console Consumer

C:\>kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --topic myfirst

We can only listen messages when the consumer is only in active state

Apache Kafka allows it to produce millions of messages. Sometimes, a consumer may require to read whole messages from a particular topic.

To do so, use '**-from-beginning**' command with the above kafka console consumer command as:

C:\>kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --topic myfirst

--from-beginning

First Producer

Hello

ratna

hyd

ratna

hyd

Ratna Created it

27

27

# Kafka Consumer Group CLI

C:\kafka>kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --topic my

first --group first\_app

No messages will come as we are having no group with name first\_app

C:\>kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --topic myfirst

--from-beginning --group first\_app

We will get previous messages

We can create more consumers for a group and the data will be distributed among the consumers equally as we have partitioned them

To get consumer groups list

C:\kafka>kafka-consumer-groups.bat --bootstrap-server 127.0.0.1:9092 --list

first\_app

Describe consumer group

C:\kafka>kafka-consumer-groups.bat --bootstrap-server 127.0.0.1:9092 --describe

--group first\_app

GROUP TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG

CONSUMER-ID HOST

CLIENT-ID

first\_app myfirst 0 4 4 0

consumer-first\_app-1-ce80c9be-6553-4e2b-a50c-edfdf87449d0 /192.168.99

.1 consumer-first\_app-1

first\_app myfirst 1 6 6 0

consumer-first\_app-1-ce80c9be-6553-4e2b-a50c-edfdf87449d0 /192.168.99

.1 consumer-first\_app-1

first\_app myfirst 2 3 3 0

consumer-first\_app-1-ce80c9be-6553-4e2b-a50c-edfdf87449d0 /192.168.99

.1 consumer-first\_app-1

For further commands use refer

<https://www.javatpoint.com/kafka-consumer-group-cli>