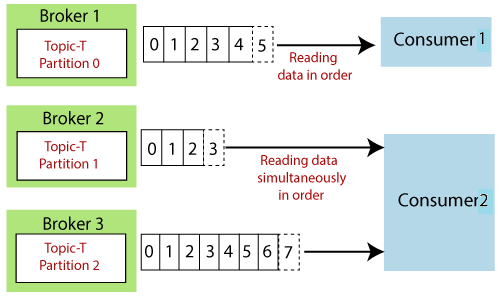
Kafka: Consumer and Consumer Groups

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.



**Note:** Consumer 2 is reading data parallelly from Broker 2 and Broker 3. Thus, offset 2 under Broker 2 does not have any connection with the data contained in offset 2 under Broker 3.

1. Consumer Groups

A consumer group is a group of multiple consumers which visions to an application basically. Each consumer present in a group reads data directly from the exclusive partitions. In case, the number of consumers are 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 takeover and will come in an active state to read the data.

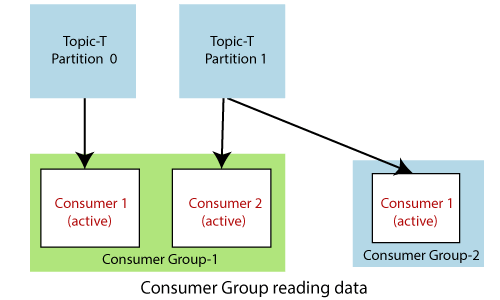
But, how to decide which consumer should read data first and from which partition?

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 the Kafka. Therefore, the user does not need to worry about it.

Let's see the below examples.

**Example 1**

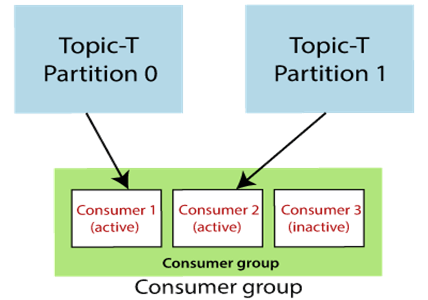
Consider two groups of consumers, i.e., Consumer Group-1 and Consumer Group-2. Both the consumers of Group 1 are reading data together but from different partitions. Both the consumers of Group 1 will remain in an active state because they are reading the data parallelly.



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 consumer leaves.



**Note: In Example 2, three consumers are present in one group only. That's why Consumer 3 is inactive. However, if the consumer is present in another group, it will be in an active state and able to read the data.**

## **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.



## **Delivery semantics**

The choice of commitment depends on the consumer, i.e., when the consumer wishes to commit the offsets. Committing an offset is like a bookmark which a reader uses while reading a book or a novel.

In Kafka, there are following three delivery semantics used:

* **At most once:** Here, the offsets are committed as soon as the consumer receives the message.. But in case of incorrect processing, the message will be lost, and the consumer will not be able to read further. Therefore, this semantic is the least preferred one.
* **At least once:** Here, the offsets are committed after the message has been processed. If the processing goes wrong, then the message will be read again by the consumer. Therefore, this is usually preferred to use. Because a consumer can read the message twice, it results in duplicate processing of the messages. Thus, it needs a system to be an idempotent system.
* **Exactly once:** Here, the offsets can be achieved for Kafka to Kafka workflow only using the Kafka Streams API. For achieving offset for Kafka to the external system, we need to use an idempotent consumer.