

Kafka and Messaging Systems



Kafka is a **distributed Message streaming platform** that uses **publish and subscribe** mechanism to stream the records.

Originally developed by LinkedIn and later donated to apache foundation.

Kafka is **open source**.

Currently used by many big enterprises like LinkedIn, Airbnb, Netflix, uber, Walmart.

MESSAGING SYSTEMS

A **messaging system** is responsible for transferring data from one application to another so the applications can focus on data without getting bogged down on data transmission and sharing.

Two types –

1. Point to Point Messaging System

1. Messages are persisted in a Queue.
2. A particular message can be consumed by a maximum of one receiver only.
3. There is no time dependency laid for the receiver to receive the message.
4. When the Receiver receives the message, it will send an acknowledgement back to the Sender.

2. Publish-Subscribe Messaging System

1. Messages are persisted in a Topic.
2. A particular message can be consumed by any number of consumers.
3. There is time dependency laid for the consumer to consume the message.
4. When the Subscriber receives the message, it doesn't send an acknowledgement back to the Publisher.

Explain the given diagram in details



The provided images appear to contain information on **Apache Kafka** and **messaging systems**.

Image 1: Apache Kafka Overview

This diagram explains **Apache Kafka** as a distributed message streaming platform with the following key points:

1. Definition:

- Kafka is a **distributed message streaming platform**.
- It employs a **publish-subscribe mechanism** to stream records.

2. Origin:

- Originally developed by **LinkedIn**.
- Later donated to the **Apache Software Foundation**.

3. Open Source:

- Kafka is **open source**, meaning anyone can use, modify, or distribute it without proprietary restrictions.

4. Usage:

- Kafka is adopted by major enterprises such as **LinkedIn, Airbnb, Netflix, Uber, and Walmart** for handling high-scale, real-time data streaming needs.

Image 2: Messaging Systems Overview

This diagram introduces **messaging systems** and their two main types:

1. Definition of Messaging System:

- A messaging system is responsible for **transferring data** between applications.
- It allows applications to **focus on their data** without worrying about transmission challenges.

2. Types of Messaging Systems:

• Point-to-Point Messaging System:

- Messages are **persisted in a queue**.
- A **particular message** can be consumed by **one receiver only**.
- There is **no time dependency**, meaning the receiver can consume the message whenever it is ready.
- Once the message is received, the receiver **sends an acknowledgment** back to the sender.

• Publish-Subscribe Messaging System:

- Messages are **persisted in a topic**.
- A **particular message** can be consumed by **multiple consumers**.
- There is a **time dependency**, as the consumer may miss messages if it is not actively subscribed.

- No acknowledgment is sent back to the publisher once the subscriber receives the message.

Connection Between Kafka and Messaging Systems

Apache Kafka operates on the **publish-subscribe model**, making it a **high-throughput and scalable messaging system**. It is particularly suited for real-time streaming use cases due to its distributed architecture and ability to handle high volumes of data efficiently.



TOPICS

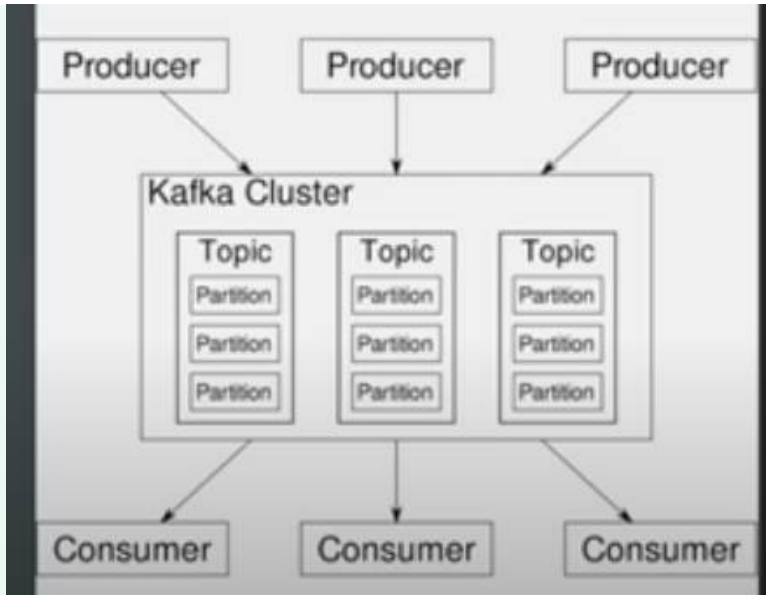
A stream of messages belonging to a particular category is called a topic.

It is a logical feed name to which records are published.

Similar to a table in a database. (records are considered messages here.)

Unique identifier of a topic is its **NAME**.

We can create as many topics as we want.



Kindly explain the given above images in details



Image 1: Kafka Cluster Architecture

This diagram showcases the internal structure of an Apache Kafka cluster, including producers, brokers, topics, partitions, and consumer groups.

1. Producers:

- These are entities that generate and send data (messages) to Kafka.
- In the diagram, three producers are shown: Producer 1, Producer 2, and Producer 3.

2. Kafka Cluster:

- The core of Kafka, consisting of **brokers**. Each broker handles message storage and retrieval.
- This Kafka cluster has three brokers (Broker 1, Broker 2, and Broker 3).

3. Topics:

- Topics act as categories or feeds to which producers send messages.
- A topic is split into **partitions** (p0, p1, p2, etc.), which are spread across brokers for scalability and fault tolerance.

4. Partitions:

- Each topic is divided into smaller segments called partitions.
- Each partition is managed by a broker and has a unique ID (e.g., p0, p1).

5. Consumers and Consumer Groups:

- Consumers retrieve messages from Kafka topics.
- They are grouped into **consumer groups** to coordinate consumption. In the diagram:
 - Group G1 has two consumers (Consumer 1 and Consumer 2).
 - Group G2 has one consumer (Consumer 3).
 - Group G3 has three consumers (Consumer 4, Consumer 5, and Consumer 6).
- Each consumer in a group reads from exclusive partitions to avoid duplicate processing.

6. Zookeeper Cluster:

- Kafka relies on Zookeeper for managing brokers, topics, and partitions. It ensures cluster health and handles leader election.

Image 2: Kafka Topics Overview

This slide explains **Kafka topics**.

1. Definition:

- A Kafka **topic** is a stream of messages belonging to a particular category.
- Producers publish messages to a topic, and consumers subscribe to topics to read data.

2. Comparison to Databases:

- Topics are similar to tables in a database, where each record is a message.

3. Unique Identifier:

- Each topic has a **unique name**.

4. Flexibility:

- Kafka allows you to create as many topics as needed, enabling flexible message categorization.

Image 3: Producer-Topic-Consumer Flow

This diagram provides an overview of how producers, topics, and consumers interact in a Kafka cluster.

1. Producers:

- Multiple producers can send messages to one or more topics.

2. Kafka Cluster:

- The Kafka cluster consists of topics, each split into **partitions**.
- Partitions allow Kafka to handle high-throughput scenarios by enabling parallel processing.

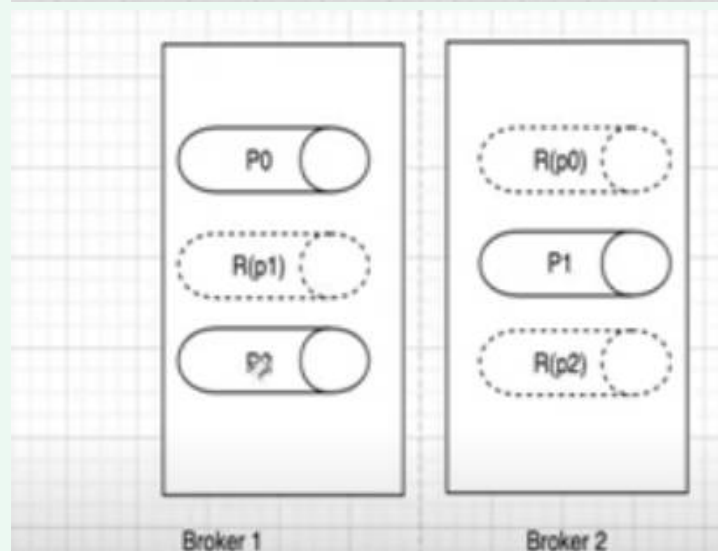
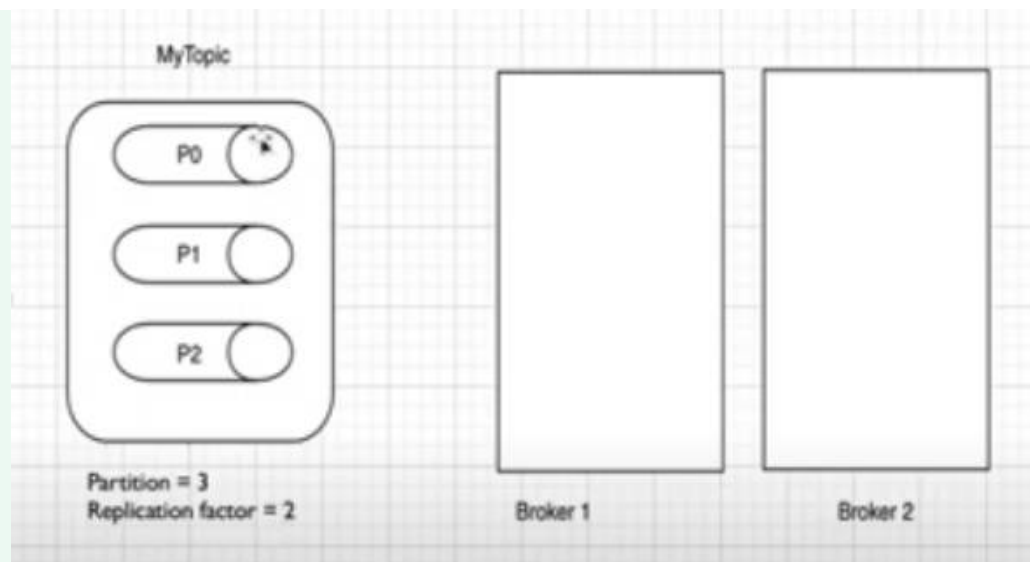
3. Consumers:

- Multiple consumers can subscribe to a topic.
- Consumers can belong to **consumer groups**, and Kafka assigns partitions to consumers within a group.

Summary Across All Images:

- Kafka is a distributed, fault-tolerant message broker that allows for scalable data processing.
- Producers send messages to **topics**, which are divided into partitions and managed by brokers.
- Consumers read messages from partitions, coordinated through **consumer groups**.
- Zookeeper is used for cluster management tasks like metadata storage and leader election.





Kindly explain the above images in details



Image 1: Replica and Replication in Kafka

1. Definition of Replicas:

- Replicas are **backups of a partition** in Kafka.
- Each partition in a Kafka topic has one or more replicas depending on the **replication factor**.

2. Purpose:

- Replicas are **never directly read or written to**; their primary role is **fault tolerance**.
- They prevent **data loss** in case the broker hosting the partition leader goes down.

3. Fault Tolerance:

- By maintaining replicas across different brokers, Kafka ensures that even if one broker fails, the data is not lost, and a new leader can be elected from the replicas.

Image 2: Partition and Replication Factor Example

1. Topic Configuration:

- The topic "MyTopic" is configured with:
 - **3 partitions** (P0, P1, P2).
 - **Replication factor = 2**, meaning each partition will have one additional replica.

2. Brokers:

- The Kafka cluster contains two brokers: **Broker 1** and **Broker 2**.
- Partitions and their replicas are distributed across these brokers to ensure fault tolerance.

Image 3: Partition and Replica Distribution

1. Leader and Replica Distribution:

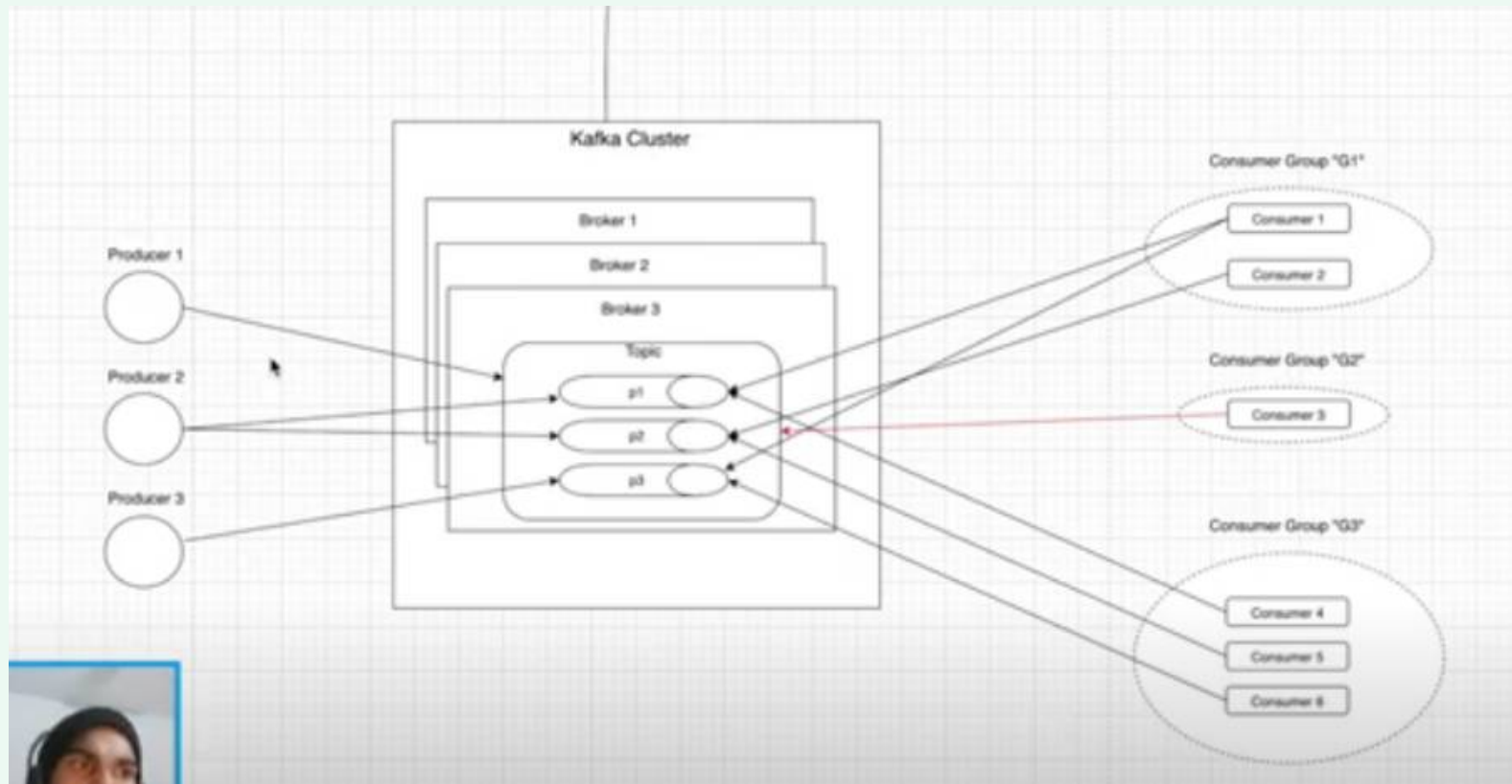
- Each partition (P0, P1, P2) has a **leader** and **replica** distributed across brokers.
- **Broker 1:**
 - Hosts the leader for **P0** and **P2**.
 - Holds the replica for **P1**.
- **Broker 2:**
 - Hosts the leader for **P1**.
 - Holds replicas for **P0** and **P2**.

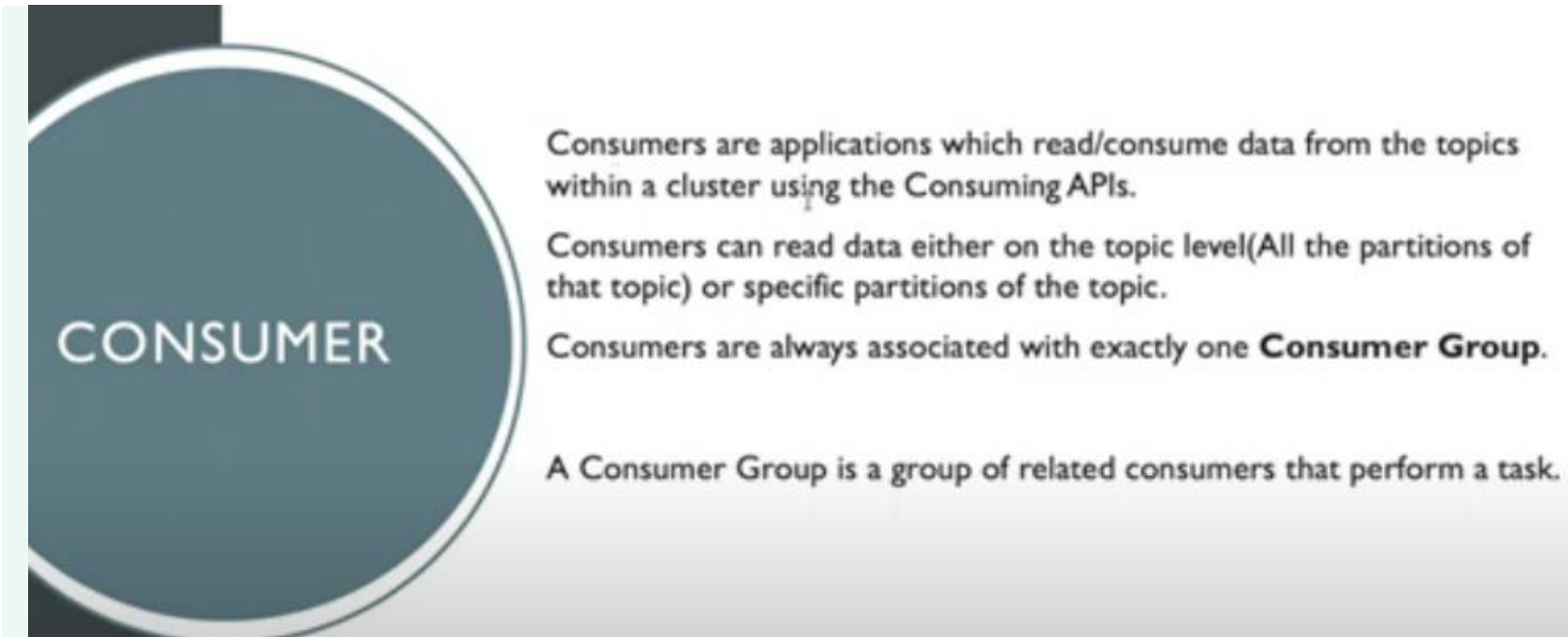
2. Replication Design:

- This ensures that if one broker fails, the other broker has the replica of the lost partition. Kafka can then elect a replica as the new leader to ensure continued availability.

Summary Across All Images

- Kafka uses **replicas** to ensure data durability and fault tolerance.
- A **replication factor** specifies how many total copies (original + replicas) exist for each partition.
- Kafka distributes partitions and their replicas across brokers for resilience, allowing seamless operation even in the event of broker failures.





Kindly explain the above images with respect to how messages are being transferred. Also explain the significance of consumer group as shown above



Explanation of Images and Message Transfer

Image 1: Kafka Message Flow with Producers and Consumers

This diagram illustrates the flow of messages in a Kafka system, including **producers**, **Kafka cluster**, **topics**, **partitions**, and **consumer groups**.

1. Producers:

- Producers are entities (applications, services, etc.) that send messages to Kafka topics.
- In this case, **Producer 1**, **Producer 2**, and **Producer 3** are sending messages to a Kafka topic with three partitions: **p1**, **p2**, **p3**.
- Each producer can send data to any partition of the topic, and Kafka handles distribution based on the key or round-robin mechanism.

2. Kafka Cluster:

- The Kafka cluster contains multiple brokers, which store partitions of topics.
- Partitions (e.g., **p1**, **p2**, **p3**) are distributed across brokers to ensure scalability and fault tolerance.

3. Topics and Partitions:

- Messages are organized into **topics**, and each topic is divided into **partitions**.
- Each partition stores a subset of the topic's messages, enabling parallelism and scalability.

4. Consumer Groups:

- Consumers are organized into **consumer groups** to efficiently consume messages:
 - **Consumer Group G1**: Two consumers (**Consumer 1 and Consumer 2**) consuming messages from different partitions.
 - **Consumer Group G2**: One consumer (**Consumer 3**) consuming messages from all partitions.
 - **Consumer Group G3**: Three consumers (**Consumer 4, Consumer 5, Consumer 6**) sharing the partitions.

Image 2: Role of Consumers and Consumer Groups

1. Consumers:

- Applications that **read/consume messages** from Kafka topics using Kafka's consuming APIs.
- Consumers can read data either from:
 - **All partitions of a topic.**
 - **Specific partitions.**

2. Consumer Groups:

- A **consumer group** is a logical grouping of one or more consumers that work together to consume data from a topic.
- Each consumer within a group consumes messages from **exclusive partitions**. No two consumers in the same group consume the same partition, ensuring no duplication of work.
- Consumers from different consumer groups can consume the same data independently.

Message Transfer Process

1. Producers:

- Send messages to a Kafka topic. Each message is assigned to a specific partition based on a key or round-robin strategy.

2. Kafka Cluster:

- Brokers store the partitions and ensure data is durably stored.
- Each partition can have a **leader** (for read/write operations) and **replicas** (for fault tolerance).

3. Consumers and Consumer Groups:

- Consumers in a group divide the workload, each processing data from specific partitions.

- Messages are consumed only once by a consumer in the group, ensuring efficient processing without duplication.
- Multiple consumer groups can process the same data independently for different use cases (e.g., analytics, storage, etc.).

Significance of Consumer Groups

1. Parallelism:

- Each consumer in a group processes messages from different partitions, enabling distributed and parallel message consumption.

2. Scalability:

- Adding more consumers to a group increases the ability to handle higher data volumes.

3. Fault Tolerance:

- If a consumer in a group fails, the partitions it was processing are reassigned to other consumers in the group.

4. Independence:

- Multiple consumer groups can consume the same messages without interfering with each other. For example:
 - One group may process data for real-time analytics.
 - Another group may store data in a database for later use.

This combination of **Kafka topics, partitions, and consumer groups** ensures efficient, scalable, and fault-tolerant message processing in a distributed system.



What is the significance if Producer forwards message to a topic and if it forwards to a specific partition



The **significance of a producer forwarding a message to a topic versus forwarding it to a specific partition** lies in how the message is distributed, stored, and consumed within the Kafka system. Here's a detailed explanation of the two scenarios:

1. Producer Forwards Message to a Topic

- When a producer sends a message to a **topic** without specifying a partition, Kafka determines the partition assignment using one of these mechanisms:
 - **Round-robin distribution:** Messages are evenly distributed across all partitions of the topic.

- **Key-based partitioning:** If a key is provided, Kafka uses a hashing function to map the message to a specific partition.
- **Default partition:** If no key or partition is specified, Kafka uses a default mechanism (usually round-robin).

Significance:

- **Even Load Distribution:**
 - Ensures that messages are evenly distributed across all partitions, which balances the workload among consumers in a consumer group.
- **No Producer Control Over Partitioning:**
 - The producer does not control which partition the message ends up in, relying entirely on Kafka's internal mechanisms.
- **Good for Stateless Applications:**
 - Ideal for cases where messages do not have interdependencies, and processing order is not critical.

2. Producer Forwards Message to a Specific Partition

- When a producer specifies a **specific partition**, all messages are sent directly to that partition.

Significance:

- **Ordered Processing:**
 - Messages sent to the same partition are processed in the **exact order** they are written. This is critical for use cases requiring message ordering (e.g., financial transactions or logs).
- **Partition Affinity:**
 - A producer can ensure that related messages (e.g., based on a specific customer ID or transaction type) are always sent to the same partition. This helps consumers process logically related data together.
- **Custom Partitioning Logic:**
 - Producers can implement custom logic to determine which partition to send the message to. For example:
 - Send high-priority messages to specific partitions for faster processing.
 - Group messages by customer, region, or other attributes.

Key Differences

Feature	Message to Topic	Message to Specific Partition
Load Distribution	Evenly distributed across partitions (by Kafka).	Messages go only to the specified partition.

Feature	Message to Topic	Message to Specific Partition
Control	Producer has no control over partitioning.	Producer explicitly selects the partition.
Use Cases	General-purpose data streaming.	Applications requiring message ordering or grouping.
Message Ordering	Ordering is guaranteed within each partition , but not across all partitions.	Ordering is guaranteed for all messages sent to that partition.
Consumer Workload	Kafka balances partitions among consumers.	Consumers handling the specified partition get all related messages.

Use Cases

Forwarding to a Topic:

- **General streaming applications** where message order is not critical.
- **Event logging** and **metrics collection**.
- Applications where data can be processed independently by consumers.

Forwarding to a Specific Partition:

- **Financial transactions** where message order is critical (e.g., debit before credit).
- **Session-based processing**, where all actions for a user session must be handled together.
- Use cases requiring **partition affinity** to group related data for efficient processing.

Conclusion

- Sending a message to a topic without specifying a partition is ideal for **general-purpose, stateless applications** where Kafka's load balancing suffices.
- Specifying a partition is crucial for **stateful, order-sensitive applications** that require tighter control over message grouping and processing.



Brokers are simple software processes who maintain and manage the published messages.

Also known as kafka servers.

Brokers also manage the **consumer-offsets** and are responsible for the delivery of messages to the right consumers.

A Set of brokers who are communicating with each other to perform the management and maintenance task are collectively known as Kafka Cluster.

We can add more brokers in a already running kafka cluster without any downtime.

Zookeeper is used to monitor kafka Cluster and co-ordinate with each broker.

Keeps all the metadata information related to kafka cluster in the form of a key-value pair.

Metadata includes

1. Configuration information.
2. Health status of each broker.

It is used for the controller election within kafka cluster.

A Set of Zookeepers nodes working together to manage other distributed systems is known as Zookeeper Cluster or "**Zookeeper Ensemble**".

Kindly explain the above images in details



Explanation of the Two Images

Image 1: Kafka Brokers

1. What are Brokers?

- Brokers are **Kafka servers** responsible for:
 - Managing and storing published messages from producers.
 - Delivering messages to the appropriate consumers.
- They act as the backbone of a Kafka cluster.

2. Managing Consumer Offsets:

- Brokers manage **consumer offsets**, which track how much of the data a consumer has processed from a partition.
- This ensures that consumers can resume consumption from the last processed offset if interrupted.

3. Kafka Cluster:

- A **Kafka cluster** is a set of brokers working together.
- These brokers communicate to manage data and perform maintenance tasks (e.g., partition management, leader election, replication).

4. Scalability:

- Kafka allows **brokers to be added dynamically** to an existing cluster without causing downtime.
- This makes Kafka highly scalable and suitable for handling growing workloads.

Image 2: Zookeeper in Kafka

1. What is Zookeeper?

- **Zookeeper** is a coordination service used by Kafka to:
 - **Monitor** the Kafka cluster.
 - **Coordinate** activities between brokers.
- It stores all metadata about the Kafka cluster in a **key-value format**.

2. Role of Zookeeper:

- Maintains critical metadata such as:
 1. **Configuration Information:**
 - Details of brokers, topics, and partitions.
 2. **Health Status:**
 - Tracks the status of each broker in the cluster.
 - Facilitates **controller election**, where one broker is elected as the cluster controller to handle administrative tasks (e.g., leader election for partitions).
- 3. Zookeeper Ensemble:**
- A **Zookeeper Ensemble** is a set of Zookeeper nodes working together to manage the Kafka cluster.
 - The ensemble ensures high availability and fault tolerance for the coordination service.

Summary of Roles

Component	Role
Kafka Broker	Manages message storage, delivery, and offsets. Forms the core of the Kafka cluster.
Zookeeper	Provides metadata storage and coordination for Kafka, including broker monitoring and leader elections.

Key Points to Note:

1. **Brokers** handle actual data storage and transfer, while **Zookeeper** is responsible for metadata and coordination.
2. Kafka is designed to run without Zookeeper in the future using a Zookeeper-less mode (e.g., Kafka's KRaft mode).
3. Zookeeper ensures the cluster remains operational even when some brokers or nodes fail, maintaining fault tolerance and high availability.