GROW DATA SKILLS

What is Kafka?

Apache Kafka is a fast, scalable, fault-tolerant messaging system which enables **communication** between **producers** and **consumers** using **message-based** topics. In simple words, it designs a platform for high-end new generation distributed applications. Before moving forward in Kafka Tutorial, let's understand the **Messaging System** in Kafka.

Messaging Systems in Kafka

The main task of managing system is to transfer data from one application to another so that the applications can mainly work on data without worrying about sharing it.

Distributed messaging is based on the reliable message queuing process. Messages are queued non-synchronously between the messaging system and client applications.

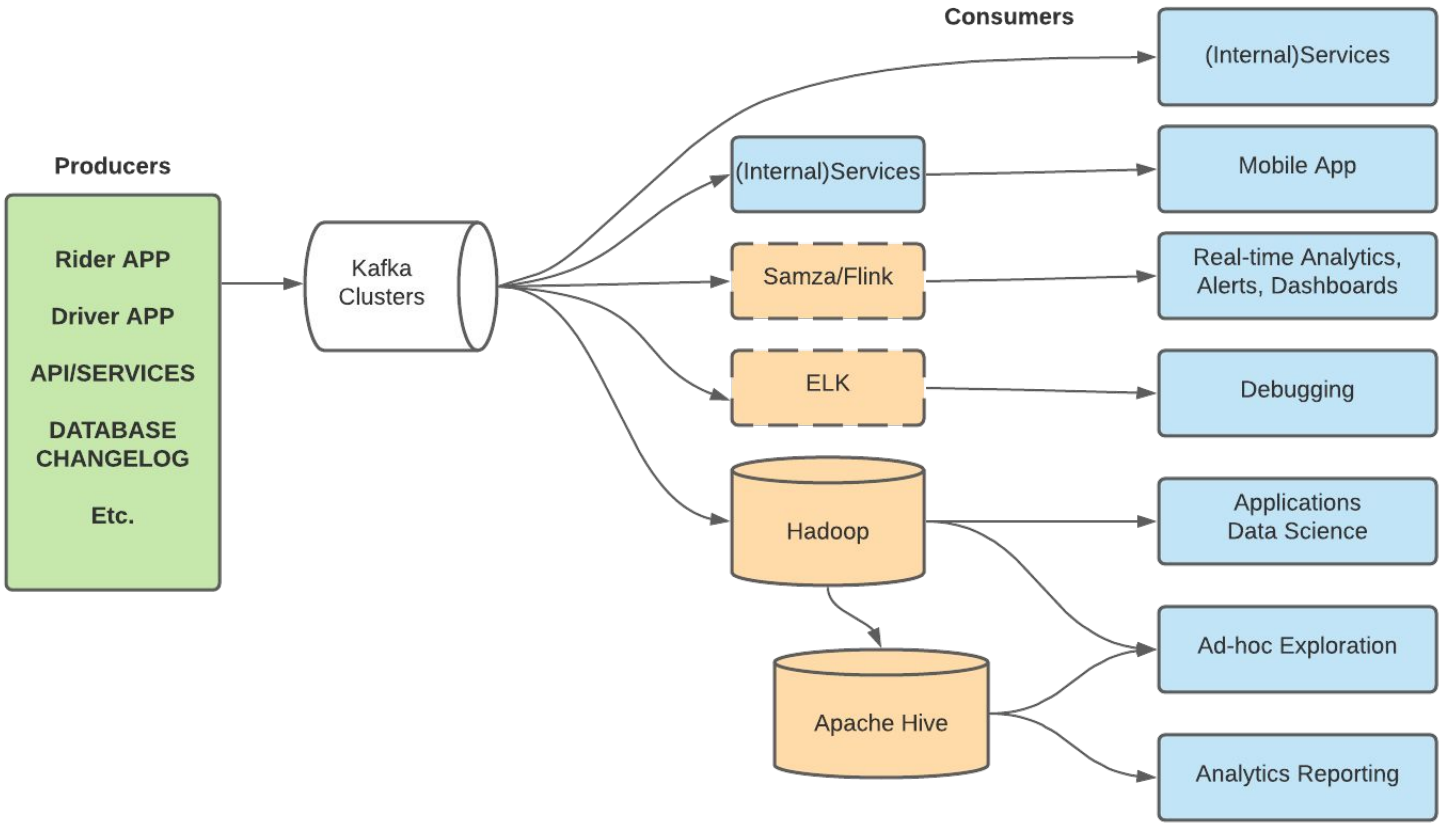
There are two types of messaging patterns available:

- Point to point messaging system
- Publish-subscribe messaging system

- **Point to Point Messaging System:** In this messaging system, messages continue to remain in a queue. More than one consumer can consume the messages in the queue but only one consumer can consume a particular message. After the consumer reads the message in the queue, the message disappears from that queue.

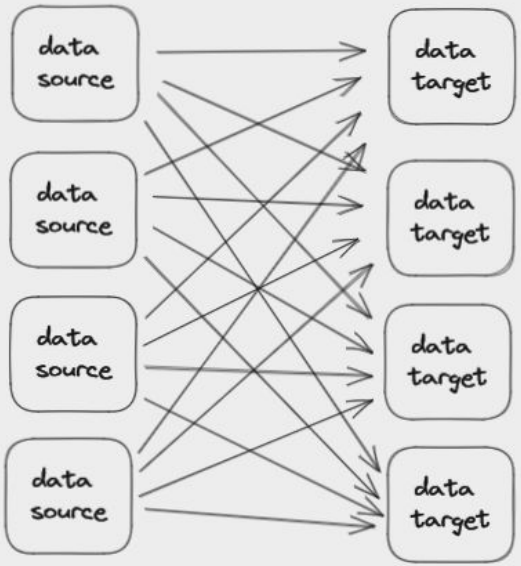
- **Publish-Subscribe Messaging System** : In this messaging system, messages continue to remain in a Topic. Contrary to Point to point messaging system, consumers can take more than one topic and consume every message in that topic. Message **producers** are known as **publishers** and Kafka **consumers** are known as **subscribers**.

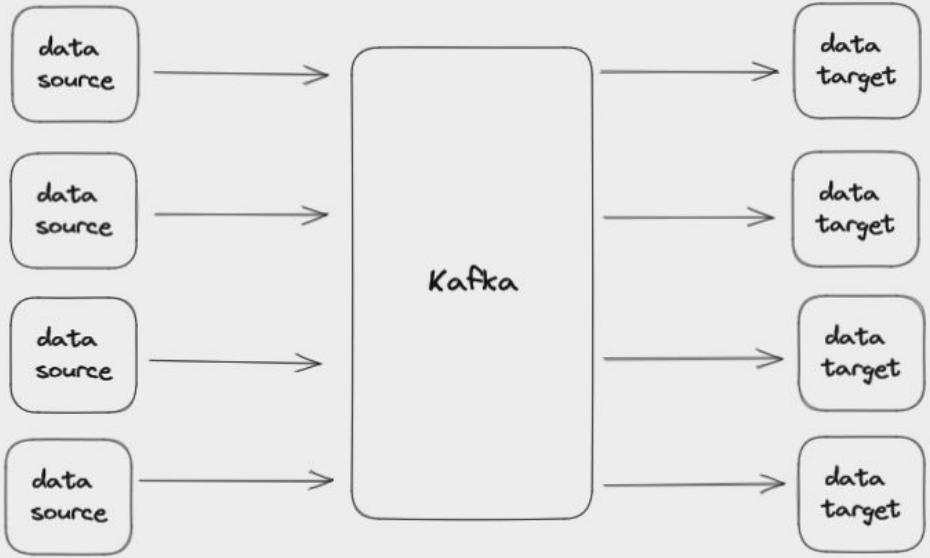
GROW DATA SKILLS



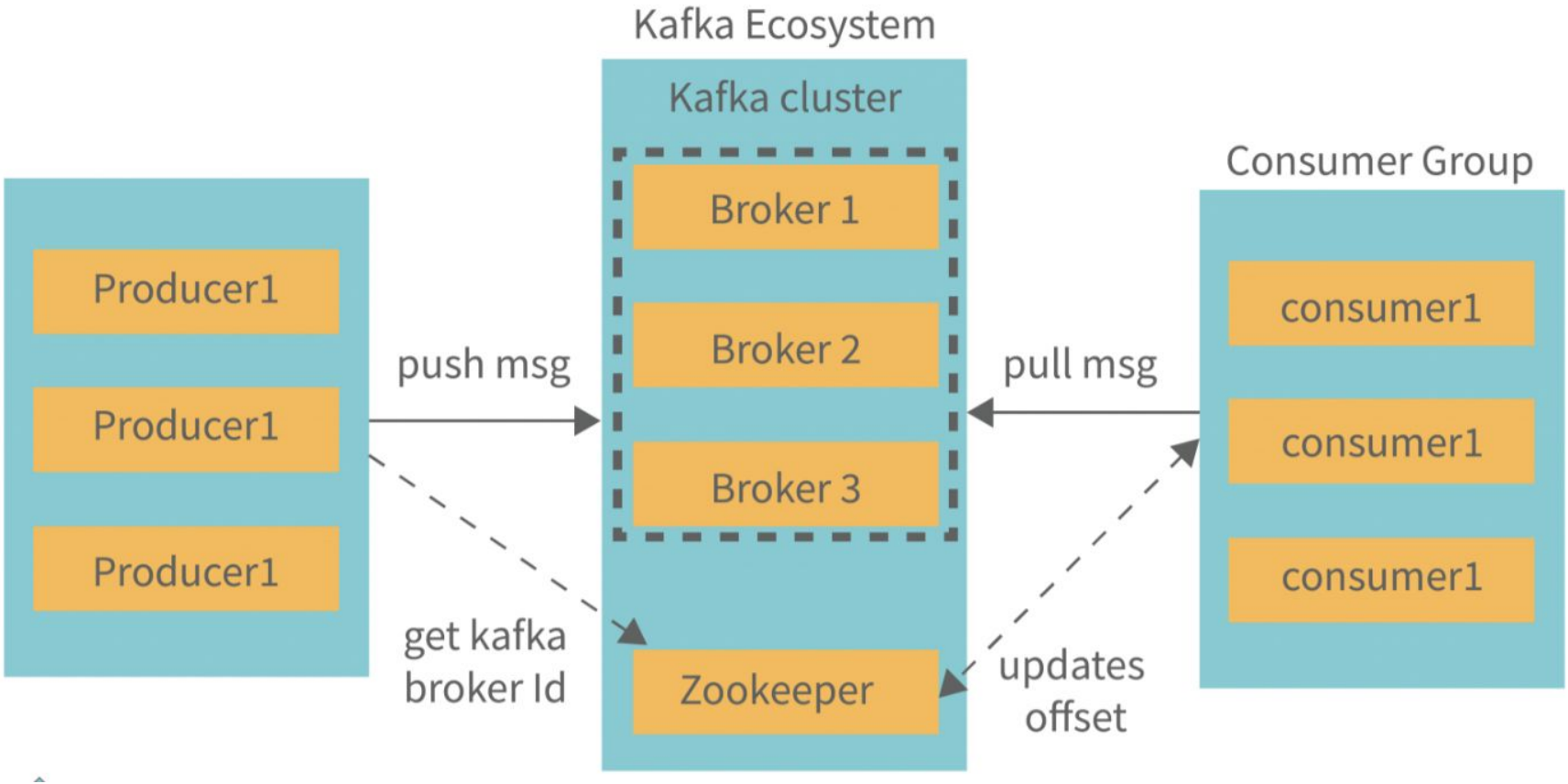
Before and After Kafka

GROW DATA SKILLS





Architecture of Kafka



Components of Kafka

- **Kafka Cluster -** A Kafka cluster is a system of multiple interconnected **Kafka brokers** (**servers**). These brokers cooperatively handle data distribution and ensure fault tolerance, thereby enabling efficient data processing and reliable storage.

- **Kafka Broker -** A Kafka broker is a server in the Apache Kafka distributed system that stores and manages the data (messages). It handles requests from producers to write data, and from consumers to read data. Multiple brokers together form a Kafka cluster.

- **Kafka Zookeeper -** Apache ZooKeeper is a service used by Kafka for cluster coordination, failover handling, and metadata management. It keeps Kafka brokers in sync, manages topic and partition information, and aids in broker failure recovery and leader election.

- **Kafka Producer -** In Apache Kafka, a producer is an application that sends messages to Kafka topics. It handles message partitioning based on specified keys, serializes data into bytes for storage, and can receive acknowledgments upon successful message delivery. Producers also feature automatic retry mechanisms and error handling capabilities for robust data transmission.

Components of Kafka

- **Kafka Consumer -** A Kafka consumer is an application that reads (or consumes) messages from Kafka topics. It can subscribe to one or more topics, deserializes the received byte data into a usable format, and has the capability to track its offset (the messages it has read) to manage the reading position within each partition. It can also be part of a consumer group to share the workload of reading messages.

What does ZooKeeper do in Kafka Cluster?

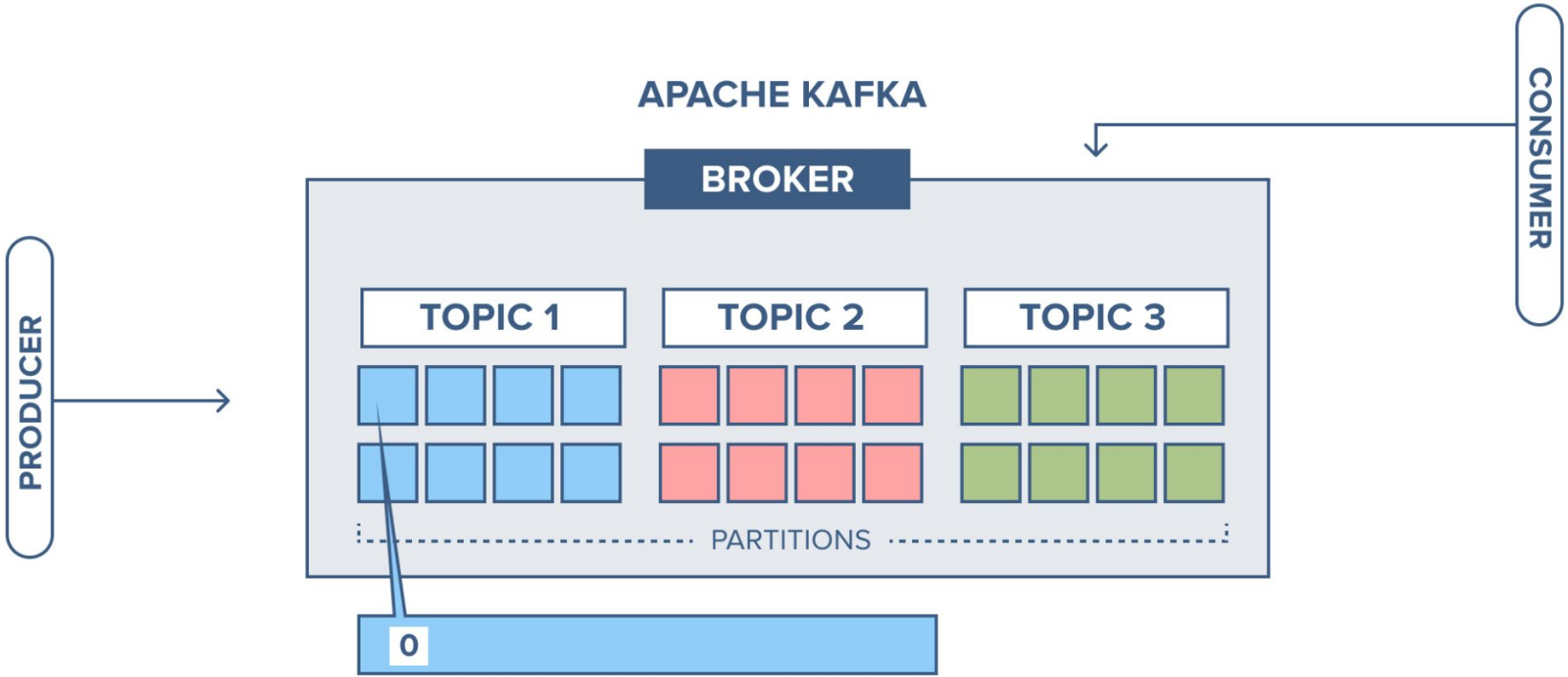
- **Broker Management**: It helps manage and coordinate the Kafka brokers, and keeps a list of them.
- **Topic Configuration Management**: ZooKeeper maintains a list of topics, number of partitions for each topic, location of each partition and the list of consumer groups.
- **Leader Election**: If a leader (the node managing write and read operations for a partition) fails, ZooKeeper can trigger leader election and choose a new leader.
- **Cluster Membership**: It keeps track of all nodes in the cluster, and notifies if any of these nodes fail.
- **Synchronization**: ZooKeeper helps in coordinating and synchronizing between different nodes in a Kafka cluster.

Topics in Kafka

GROW DATA SKILLS

- **Topic:** A Kafka Topic is a category or stream name to which messages are published by the Producers and retrieved by the Consumers.
- **Partitions**: Each Kafka topic is divided into one or more partitions. Partitions allow for data to be parallelized, meaning data can be written to or read from multiple partitions at once, increasing overall throughput.
- **Producer Data Writing**: Producers write data to topics, and they typically choose which partition to write to within the topic either in a round-robin style or using a semantic partition function.
- **Consumer Data Reading**: Consumers read data from topics. They read from each partition in the order the data was written, maintaining what's known as an offset to track their progress.
- **Retention Policy**: Kafka topics retain all messages for a set amount of time, regardless of whether they have been consumed. This time period is configurable per topic.
- **Immutable Records**: Once a message is written to a Kafka topic partition, it can't be changed (it's immutable). It's available to consumers to read until

Topics in Kafka



Partitions in Kafka

GROW DATA SKILLS

In Apache Kafka, a partition is a division of a Kafka topic. Partitions play a crucial role in Kafka's functionality and scalability. Here's how:

- **Parallelism**: Partitions enable parallelism. Since each partition can be placed on a separate machine (broker), a topic can handle an amount of data that exceeds a single server's capacity. This allows producers and consumers to read and write data to a topic concurrently, thus increasing throughput.

- **Ordering**: Kafka guarantees that messages within a single partition will be kept in the exact order they were produced. However, if order is important across partitions, additional design considerations are needed.

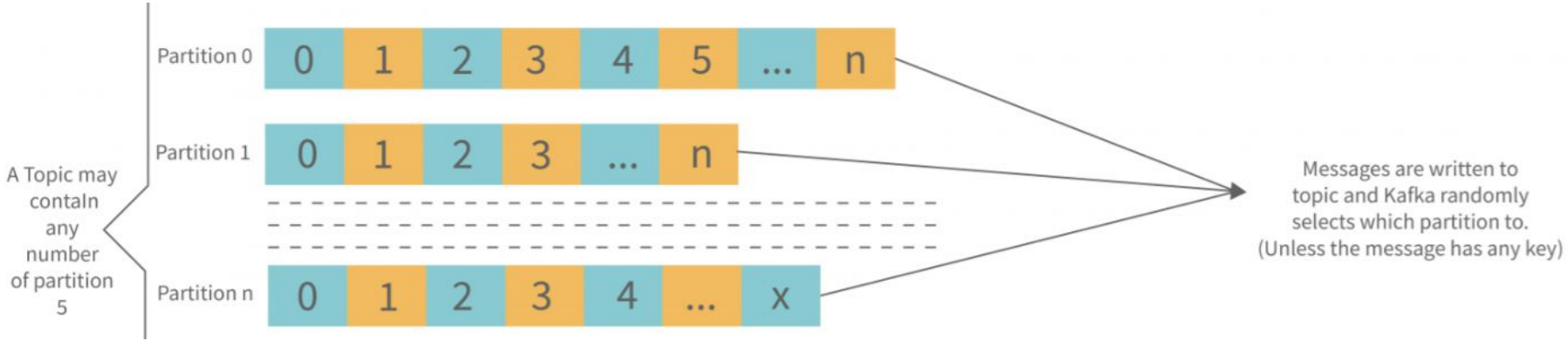
- **Replication**: Partitions of a topic can be replicated across multiple brokers based on the topic's replication factor. This increases data reliability and availability.

- **Failover**: In case of a broker failure, the leadership of the partitions owned by that broker will be automatically taken over by another broker, which has the replica of these partitions.

- **Consumer Groups**: Each partition can be consumed by one consumer within a consumer group at a time. If more than one consumer is needed to read data from a topic simultaneously, the topic needs to have more than one partition.

- **Offset**: Every message in a partition is assigned a unique (per partition) and sequential ID called an offset. Consumers use this offset to keep track of their position in the partition.

Partitions in Kafka



How Kafka Producer publishes a message in partition?

GROW DATA SKILLS

The decision of which partition a message is written to within a topic is determined by the Kafka producer using one of the following methods:

- **Round-Robin (Default)**: If no key is provided, the producer follows a round-robin method to write to the next available partition. This ensures a balance of data across all partitions.

- **Keyed Message**: If a key is provided with the message, a hash of the key is used to determine the partition. This ensures that all messages with the same key always go to the same partition, thereby maintaining order for that key.

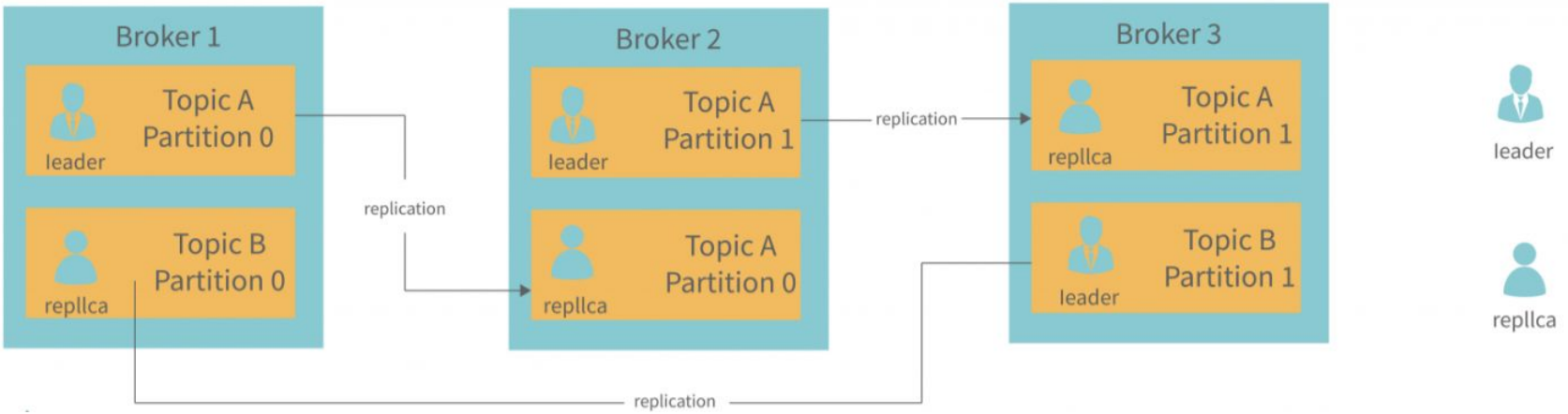
- **Custom Partitioner**: Producers can also define a custom partitioning strategy by implementing a custom partitioner. This allows more control over the partitioning process.

These strategies help ensure efficient data writing, balancing data evenly across partitions, maintaining order where necessary, and supporting custom use cases.

Topic Replication Factor in Kafka

When designing a Kafka system, it's important to include topic replication in the algorithm. When a broker goes down, its topic replicas from another broker can solve the crisis, assuming there is no partitioning. We have 3 brokers and 3 topics.

Topic 1 and Partition 0 both have a replication factor of 2, and so on and so forth. Broker1 has Topic 1 and Partition 0, and Broker2 has Broker2. It has got a replication factor of 2; which means it will have one additional copy other than the primary copy. The image is below:



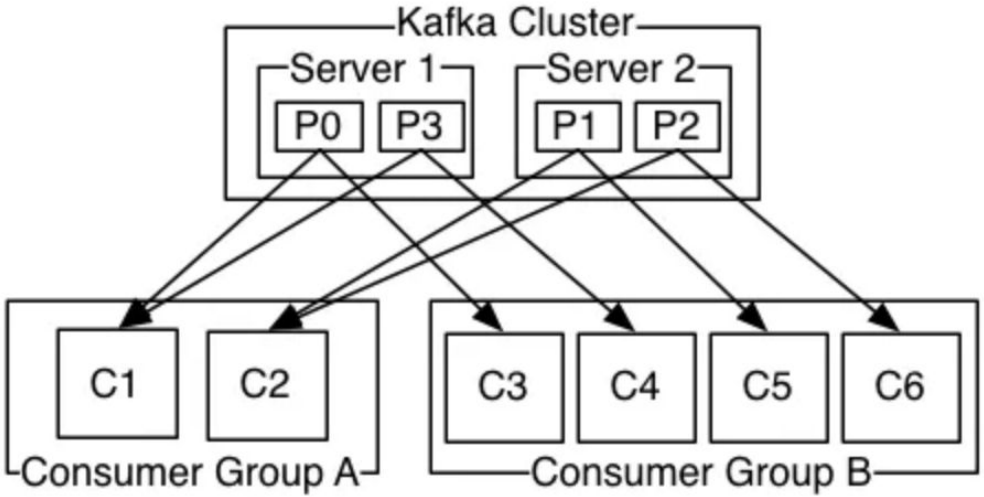
Some important points are stated:

- The level of replication performed is partition level only.
- There can be only one broker leader at a time for a given partition. Meanwhile, other brokers will maintain synchronised replicas.
- Having more than the number of brokers would result in an over-saturation of the replication factor.

Consumer Group

A consumer group is a collection of consumers who work together to consume data on a specific topic. All of the topics' partitions are divided among the group's consumers.

Kafka's consumer groups allow it to take advantage of both message queuing and publish-subscribe architectures. A group id is shared by Kafka consumers who belong to the same consumer group. The consumers in a group then distribute the topic partitions as evenly as feasible amongst themselves by ensuring that each partition is consumed by just one consumer from the group.



How Consumers in Consumer Group read messages?

GROW DATA SKILLS

- A single Kafka consumer can read from all partitions of a topic. This is often the case when you have only one consumer in a consumer group.

- However, when you have multiple consumers in a consumer group, the partitions of a topic are divided among the consumers. This allows Kafka to distribute the data across the consumers, enabling concurrent data processing and improving overall throughput.

- It's also important to note that while a single consumer can read from multiple partitions, a single partition can only be read by one consumer from a specific consumer group at a time. This ensures that the order of the messages in the partition is maintained when being processed.

Advantages of Consumer Group

GROW DATA SKILLS

Consumer groups in Apache Kafka have several key advantages:

- **Load Balancing**: Consumer groups allow the messages from a topic's partitions to be divided among multiple consumers in the group. This effectively balances the load and allows for higher throughput.
- **Fault Tolerance**: If a consumer in a group fails, the partitions it was consuming from are automatically reassigned to other consumers in the group, ensuring no messages are lost or left unprocessed.
- **Scalability**: You can increase the processing speed by simply adding more consumers to a group. This makes it easy to scale your application according to the workload.
- **Parallelism**: Since each consumer in a group reads from a unique set of partitions, messages can be processed in parallel, improving the overall speed and efficiency of data processing.
- **Ordering Guarantee**: Within each partition, messages are consumed in order. As a single partition is consumed by only one consumer in the group, this preserves the order of messages as they were written into the partition.

Rebalancing in Kafka

GROW DATA SKILLS

In Apache Kafka, rebalancing refers to the process of redistributing the partitions of topics across all consumers in a consumer group. Rebalancing ensures that all consumers in the group have an equal number of partitions to consume from, thus evenly distributing the load.

Rebalancing can be triggered by several events:

- **Addition or removal of a consumer**: If a new consumer joins a consumer group, or an existing consumer leaves (or crashes), a rebalance is triggered to redistribute the partitions among the available consumers.

- **Addition or removal of a topic's partition**: If a topic that a consumer group is consuming from has a partition added or removed, a rebalance will be triggered to ensure that the consumers in the group are consuming from the correct partitions.

- **Consumer finishes consuming all messages in its partitions**: When a consumer has consumed all messages in its current list of partitions and commits the offset back to Kafka, a rebalance can be triggered to assign it new partitions to consume from.

While rebalancing ensures fair partition consumption across consumers, it's important to note that it can also cause some temporary disruption to the consuming process, as consumers may need to stop consuming during the rebalance. To minimize the impact, Kafka allows you to control when and how a consumer commits its offset, so you can ensure it happens at a point that minimizes any disruption from a rebalance.

Offset Management in Kafka

GROW DATA SKILLS

In Apache Kafka, consumer offset management – that is, tracking what messages have been consumed – is handled by Kafka itself.

When a consumer in a consumer group reads a message from a partition, it **commits** the **offset** of that message back to Kafka. This allows Kafka to keep track of what has been consumed, and what messages should be delivered if a new consumer starts consuming, or an existing consumer restarts.

Earlier versions of Kafka used Apache ZooKeeper for offset tracking, but since version 0.9, Kafka uses an internal topic named **"__consumer_offsets"** to manage these offsets. This change has helped to improve scalability and durability of consumer offsets.

Kafka maintains two types of offsets.

1. **Current Offset** : The current offset is a reference to the most recent record that Kafka has already provided to a consumer. As a result of the current offset, the consumer does not receive the same record twice.

2. **Committed Offset** : The committed offset is a pointer to the last record that a consumer has successfully processed. We work with the committed offset in case of any failure in application or replaying from a certain point in event stream.

Offset Management in Kafka

GROW DATA SKILLS

Committing an offset

When a consumer in a group receives messages from the partitions assigned by the coordinator, it must commit the offsets corresponding to the messages read. If a consumer crashes or shuts down, its partitions will be reassigned to another member, who will start consuming each partition from the previous committed offset. If the consumer fails before the offset is committed, then the consumer which takes over its partitions will use the reset policy.

There are two ways to commit an offset:

1. **Auto Commit** : By default, the consumer is configured to use an automatic commit policy, which triggers a commit on a periodic interval. This feature is controlled by setting two properties:
    - enable.auto.commit
    - auto.commit.interval.ms

Although auto-commit is a helpful feature, it may result in duplicate data being processed.

Let’s have a look at an example.

You’ve got some messages in the partition, and you’ve requested your first poll. Because you received ten messages, the consumer raises the current offset to ten. You process these ten messages and initiate a new call in four seconds. Since five seconds have not passed yet, the consumer will not commit the offset. Then again, you’ve got a new batch of records, and rebalancing has been triggered for some reason.

Offset Management in Kafka

The first ten records have already been processed, but nothing has yet been committed. Right? The rebalancing process has begun. As a result, the partition is assigned to a different consumer. Because we don't have a committed offset, the new partition owner should begin reading from the beginning and process the first ten entries all over again.

A manual commit is the solution to this particular situation. As a result, we may turn off auto-commit and manually commit the records after processing them.

2. Manual Commit : With Manual Commits, you take the control in your hands as to what offset you'll commit and when. You can enable manual commit by setting the enable.auto.commit property to false.

There are two ways to implement manual commits :

- **Commit Sync :** The synchronous commit method is simple and dependable, but it is a blocking mechanism. It will pause your call while it completes a commit process, and if there are any recoverable mistakes, it will retry. Kafka Consumer API provides this as a prebuilt method.

- **Commit Async :** The request will be sent and the process will continue if you use asynchronous commit. The disadvantage is that commitAsync does not attempt to retry. However, there is a legitimate justification for such behavior.

  Let's have a look at an example.

Offset Management in Kafka

GROW DATA SKILLS

Assume you're attempting to commit an offset as 70. It failed for whatever reason that can be fixed, and you wish to try again in a few seconds. Because this was an asynchronous request, you launched another commit without realizing your prior commit was still waiting. It's time to commit-100 this time. Commit-100 is successful, however commit-75 is awaiting a retry. Now how would we handle this? Since you don't want an older offset to be committed.

This could cause issues. As a result, they created asynchronous commit to avoid retrying. This behavior, however, is unproblematic since you know that if one commit fails for a reason that can be recovered, the following higher level commit will succeed.

What if AsyncCommit failure is non-retryable?

GROW DATA SKILLS

Asynchronous commits can fail for a variety of reasons. For example, the Kafka broker might be temporarily down, the consumer may be considered dead by the group coordinator and kicked out of the group, the committed offset may be larger than the last offset the broker has, and so on.

When the commit fails with a non-retryable error, the commitAsync method doesn't retry the commit, and your application doesn't get a direct notification about it, because it runs in the background. However, you can provide a **callback function** that gets triggered upon a commit failure or success, which can log the error and you can take appropriate actions based on it.

In Python, the **commit()** function allows passing a callback function, where you can handle errors.

Here is an example:

```
def commit_callback(err, offsets):
    if err is not None:
        print(f"Commit failed: {str(err)}")
    else:
        print(f"Commit succeeded with offsets: {offsets}")

# In your consumer loop
consumer.commit(asynchronous=True, callback=commit_callback)
```

What if AsyncCommit failure is non-retryable?

GROW DATA SKILLS

But keep in mind, even if you handle the error in the callback, the commit has failed and it's not retried, which means the consumer offset hasn't been updated in Kafka. The consumer will continue to consume messages from the failed offset. In such scenarios, manual intervention or alerts might be necessary to identify the root cause and resolve the issue.

On the other hand, synchronous commits (commitSync) will retry indefinitely until the commit succeeds or encounters a non-retryable failure, at which point it throws an exception that your application can catch and handle directly. This is why it's often recommended to have a final synchronous commit when you're done consuming messages.

As a general strategy, it's crucial to monitor your consumers and Kafka infrastructure for such failures and handle them appropriately to ensure smooth data processing and prevent data loss or duplication.

When to use SyncCommit vs AsyncCommit?

GROW DATA SKILLS

Choosing between synchronous and asynchronous commit in Apache Kafka largely depends on your application's requirements around data reliability and processing efficiency.

Here are some factors to consider when deciding between synchronous and asynchronous commit:

- **Synchronous commit (commitSync)**: Use it when data reliability is critical, as it retries indefinitely until successful or a fatal error occurs. However, it can block your consumer, slowing down processing speed.

- **Asynchronous commit (commitAsync)**: Use it when processing speed is important and some data loss is tolerable. It doesn't block your consumer but doesn't retry upon failures.

- **Combination**: Many applications use commitAsync for regular commits and commitSync before shutting down to ensure the final offset is committed. This approach balances speed and reliability.

Read strategies in Kafka

GROW DATA SKILLS

In Apache Kafka, the consumer's position is referred to as the "offset". Kafka maintains the record of the current offset at the consumer level and provides control to the consumer to consume records from a position that suits their use case. This ability to control where to start reading records provides flexibility to the consumers. Here are the main reading strategies:

- **Read From the Beginning**: If a consumer wants to read from the start of a topic, it can do so by setting the consumer property ***auto.offset.reset*** to ***earliest***. This strategy is useful for use cases where you want to process all the data in the topic.

- **Read From the End (Latest)**: If a consumer only cares about new messages and doesn't want to read the entire history of a topic, it can start reading from the end. This is done by setting ***auto.offset.reset*** to ***latest***.

- **Read From a Specific Offset:** If a consumer wants to read from a particular offset, it can do so using the **seek()** method on the KafkaConsumer object. This method changes the position of the consumer to the specified offset.

- **Committing and Reading from Committed Offsets**: The consumer can commit offsets after it has processed messages. If the consumer dies and then restarts, it can continue processing from where it left off by reading the committed offset.

These strategies give consumers great flexibility in processing records in Kafka topics. Depending on your application's requirements, you can choose the one that suits you best.

How does Apache Kafka integrate into data engineering and data pipeline processes?

Apache Kafka is crucial in data engineering and data pipelines:

- **Data Ingestion & Integration**: Kafka ingests real-time data from various sources, acting as a medium for different applications to exchange data.

- **Real-Time Analytics & Stream Processing**: Kafka enables real-time data processing and analytics, transforming data as it arrives.

- **Event Sourcing & Decoupling**: Kafka provides an audit trail of events, serving as a buffer between data producers and consumers, decoupling system dependencies.

Use cases include real-time analytics, transaction processing, log aggregation, and stream processing for recommendations or IoT sensor data. Kafka's strength lies in its ability to handle real-time data in a scalable, fault-tolerant manner.

How does Apache Kafka integrate into data engineering and data pipeline processes?