APACHE KAFKA FOR DEVELOPERS

HANDBOOK

RAMA MUKKAMALLA

www.millionvisit.blogspot.com

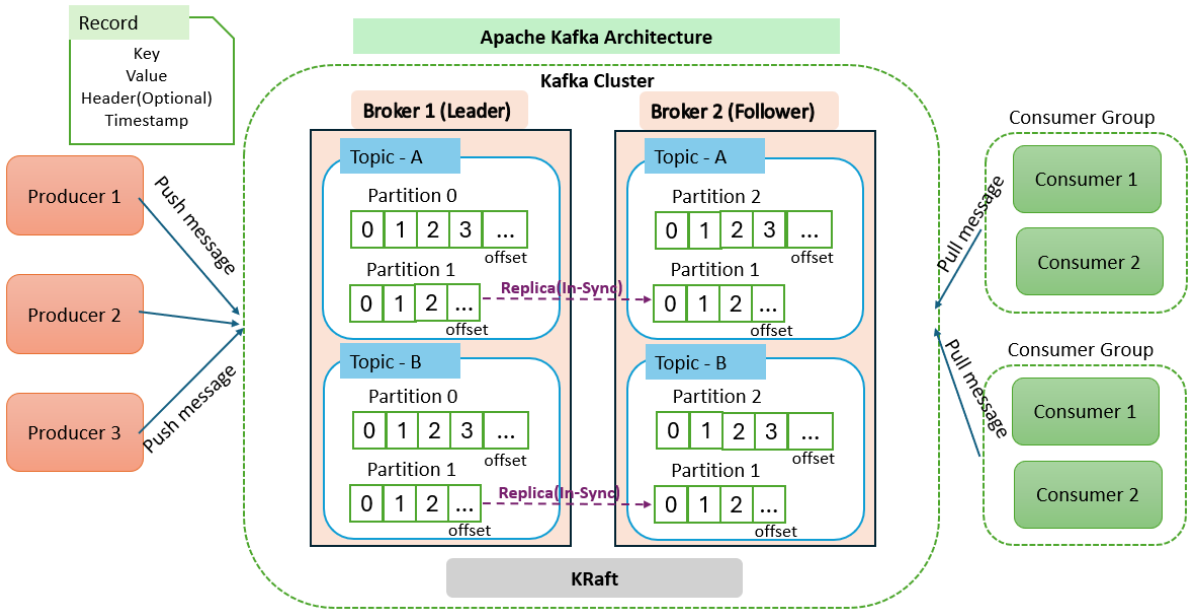
Table of Contents

1. Introduction to Kafka and Comparison with RabbitMQ

Apache Kafka is an is an open-source, distributed event streaming platform developed by LinkedIn. It is designed to handle real-time data feeds with high throughput and low latency.

It offers high throughput, fault tolerance, resilience, and scalability. It supports a range of use cases, including data integration from various data sources using data connectors, log aggregation, real-time stream processing, website activity tracking, event sourcing and publish-subscribe messaging.

Kafka's architecture is based on a distributed commit log, where data is partitioned and replicated across multiple servers to ensure fault tolerance and scalability. Producers send data to Kafka topics, which are split into partitions, and consumers read data from these partitions.



Key Characteristics of distributed commit log

1. **Append-Only**: New records are always appended to the end of the log, ensuring that the order of events is preserved.
2. **Immutable Records**: Once a record is written to the log, it cannot be changed or deleted. This immutability guarantees consistency and reliability.
3. **Sequential Reads**: Records are read in the order they were written, which simplifies the process of replaying events.
4. **Replication**: Data is replicated across multiple nodes to provide fault tolerance. If one node fails, the data can still be accessed from another node.
5. **Scalability**: By partitioning the log across multiple nodes, the system can handle large volumes of data with high throughput

Generally, two major messaging models are used to facilitate communication between the multiple applications in decoupled way includes,

Point-to-Point messaging Model:

Messages are stored in a queue, where one or more consumers can access them. However, each message can only be consumed by a single consumer. Once a consumer reads a message, it is removed from the queue.

Publish-Subscribe messaging Model:

Messages are stored in a topic. consumers can subscribe to one or more topics and consume all the messages within those topics.

Kafka's topic partitioned log architecture enables it to support both the Queuing (Point-to-Point) and Publish-Subscribe messaging models.

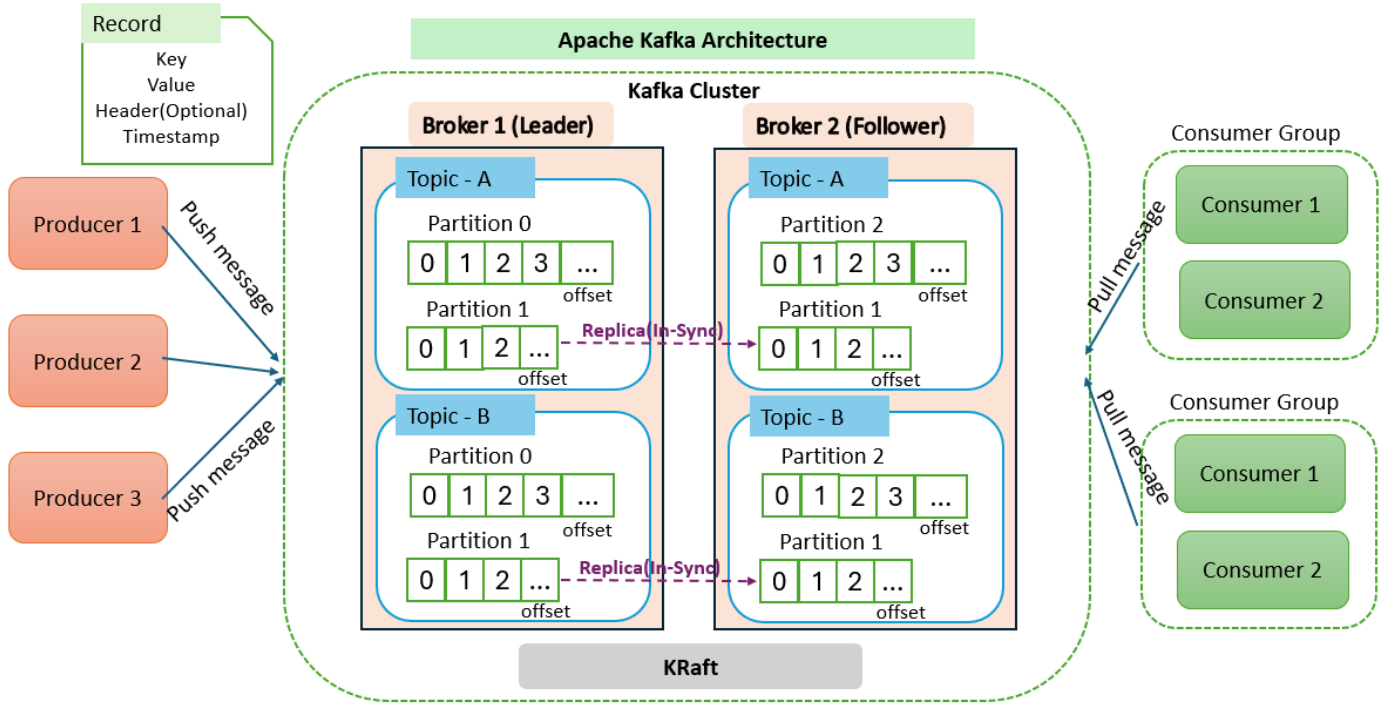
Kafka Vs RabbitMQ

Kafka	RabbitMQ
It uses a log-based architecture where messages are stored in topics. These topics are divided into partitions to ensure scalability and fault tolerance. Producers send messages to these topics, and consumers read from them at their own pace.	It uses a queue-based architecture where producers send messages to exchanges. These exchanges route the messages to queues based on routing keys, and consumers then read the messages from these queues.
It delivers high throughput and low latency, capable of handling millions of messages per second.	It delivers low latency, capable of handling thousands of messages per second.
It is ideally suited for real-time data processing, event sourcing, log aggregation, website activity tracking and stream processing.	It is ideally suited for task queues, background job processing, communication between applications and complex routing logic.
It doesn’t support publishing messages based on priority order	It supports assigning priorities to messages and consuming them based on the highest priority.
It uses a pull-based model where consumers request messages from specific offsets, enabling message replay	It uses a push-based model, delivering messages to consumers as they arrive.

and batch processing.	
Messages are stored durably according to the specified retention period.	Messages are removed once they have been consumed by the consumers.
Multiple consumers can subscribe to the same topic in Kafka, as it supports same message can be consumed by different consumers using consumer groups.	Multiple consumers cannot all receive the same message, as messages are deleted once they are consumed.
It uses a binary protocol over TCP.	It uses AMQP, STOMP and MQTT protocols

2. Kafka Architecture and Components

Apache Kafka's architecture is designed to handle high-throughput, real-time data streams efficiently and at scale. Here are the key components:



Broker

Kafka operates as a cluster composed of one or more servers, known as brokers. These brokers are

responsible for storing data and handling client requests. Each broker has a unique ID and can manage hundreds of thousands of read and write operations per second from thousands of clients.

Topic

A topic is a category where records are stored in the Kafka cluster. Topics are divided into multiple partitions, enabling parallel data processing.

It is like tables in the database.

Partition

A topic divided into multiple partitions for scalability and fault tolerance. Each partition is an ordered, immutable sequence of records that is continually appended to a commit log. Partitions allow Kafka to scale horizontally by distributing data across multiple servers.

Example

- If a topic has 3 partitions and there are 3 brokers, each broker will have one partition.
- If a topic has 3 partitions and there are 5 brokers, the first 3 brokers will each have one partition, while the remaining 2 brokers will not have any partitions for that specific topic.
- If a topic has 3 partitions and there are 2 brokers, each broker will share more than partition which leads to unequal distribution of load.

Partition Offset

Each message within a partition is assigned a unique identifier called an offset. This offset acts as a position marker for the message within the partition.

Offsets are immutable and assigned in a sequential order as messages are produced to a partition.

Consumers use offsets to keep track of their position in a partition. By storing the offset of the last consumed message, consumers can resume reading from the correct position in the event of restart or failure.

Consumers can commit offset automatically at regular intervals or commit manually after processing each message.

Consumers can also start reading from a specific offset based on application needs.

Producer

Producers are clients that publish the messages to Kafka topics. Producers send data to the broker,

which then stores it in the appropriate partition of the topic.

Consumer

Consumers are clients that read data from Kafka topics. They subscribe to one or more topics and process the data. Each consumer keeps track of its position in each partition using partition offset.

Consumer Group

A group of consumers work together to consume data from a topic. Each consumer in the group processes data from different partitions, allowing for parallel processing and load balancing.

KRaft

It is used to manage and coordinate the brokers, assisting with leader election for partitions, configuration management, and cluster metadata. In older versions of Kafka, ZooKeeper is utilized to perform these tasks.

Record

A Kafka record, also known as a message, is a unit of data in Kafka. It consists of

- **Key**: It is optional and helps determine the partition for a record. Kafka uses a hashing algorithm to map the key to a specific partition, ensuring all records with the same key go to the same partition, maintaining their order.
- **Value**: It contains actual data and can by type such as a string, JSON, or binary data, depending on the serialization format used.
- **Header**: These are optional key-value pairs that can be included with a record. They provide additional metadata about the record
- **Timestamp**: Each record has a timestamp to indicate when the record was published. This timestamp can be set by the producer or assigned by the Kafka broker when the record is received.

Partition Replicas

Each partition in a Kafka topic can have multiple replicas, which are distributed across different brokers in the cluster to ensure fault tolerance and high availability.

Leader

Each partition in a Kafka topic has a single leader. The leader is responsible for handling all read and write requests for that partition. This ensures that all data for a partition is processed in a consistent

and orderly manner.

Follower

Partitions have one or more followers. The Followers replicate the data from the leader using in-sync replicas (ISR) to ensure strong redundancy and durability.

Apache Kafka offers following five core Java APIs to facilitate cluster and client management.

1. **Producer API**: It allows applications to write stream of records to one or more Kafka topics.

2. **Consumer API**: It allows applications to read stream of records Kafka topics.

3. **Kafka Streams API**: It allows applications to read data from input topics, perform transformations, filtering, and aggregation, and then write the results back to output topics.

4. **Kafka Connect API**: It is used to develop and run reusable data connectors that pull data from external systems and send it to Kafka topics, or vice versa.

5. **Admin API**: It is used for managing Kafka clusters, brokers, topics and ACLs

3. Kafka Topic Replication

Kafka topic replication ensures data durability and high availability by duplicating each partition across multiple brokers in a Kafka cluster.

Kafka follows a leader-follower Replica model in which each partition has single leader and multiple follower replicas. The leader handles all read and write operations, while the followers replicate the data from the leader.

Replication Factor

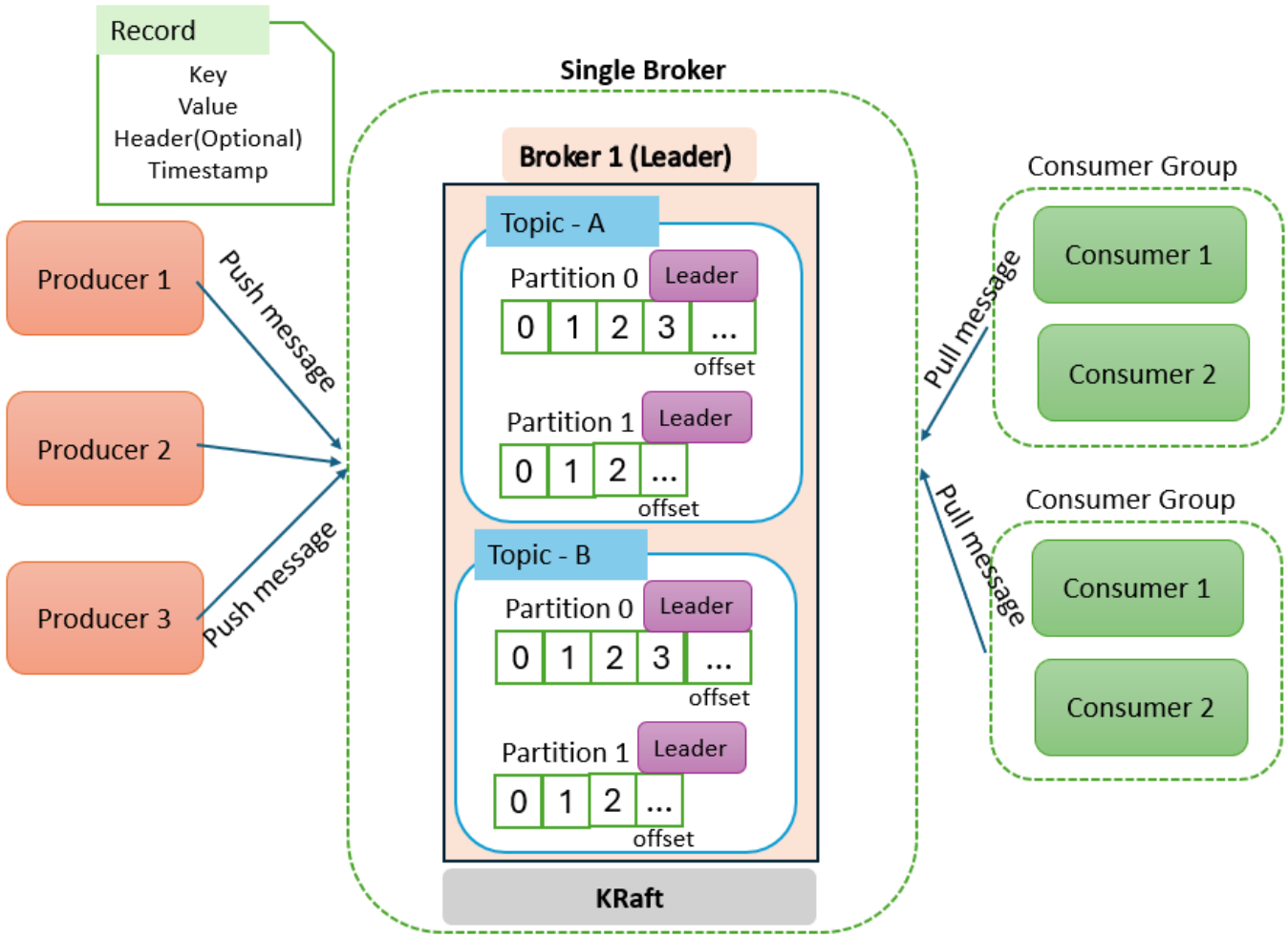
The replication factor is set at the topic level when the topic is created. It specifies how many copies of each partition will be stored across different brokers in a Kafka cluster

In-Sync Replicas (ISR)

The replicas that have fully synchronized with the leader for a specific partition are known as In-Sync Replicas. This means that all In-Sync Replicas and the leader contain the same data.

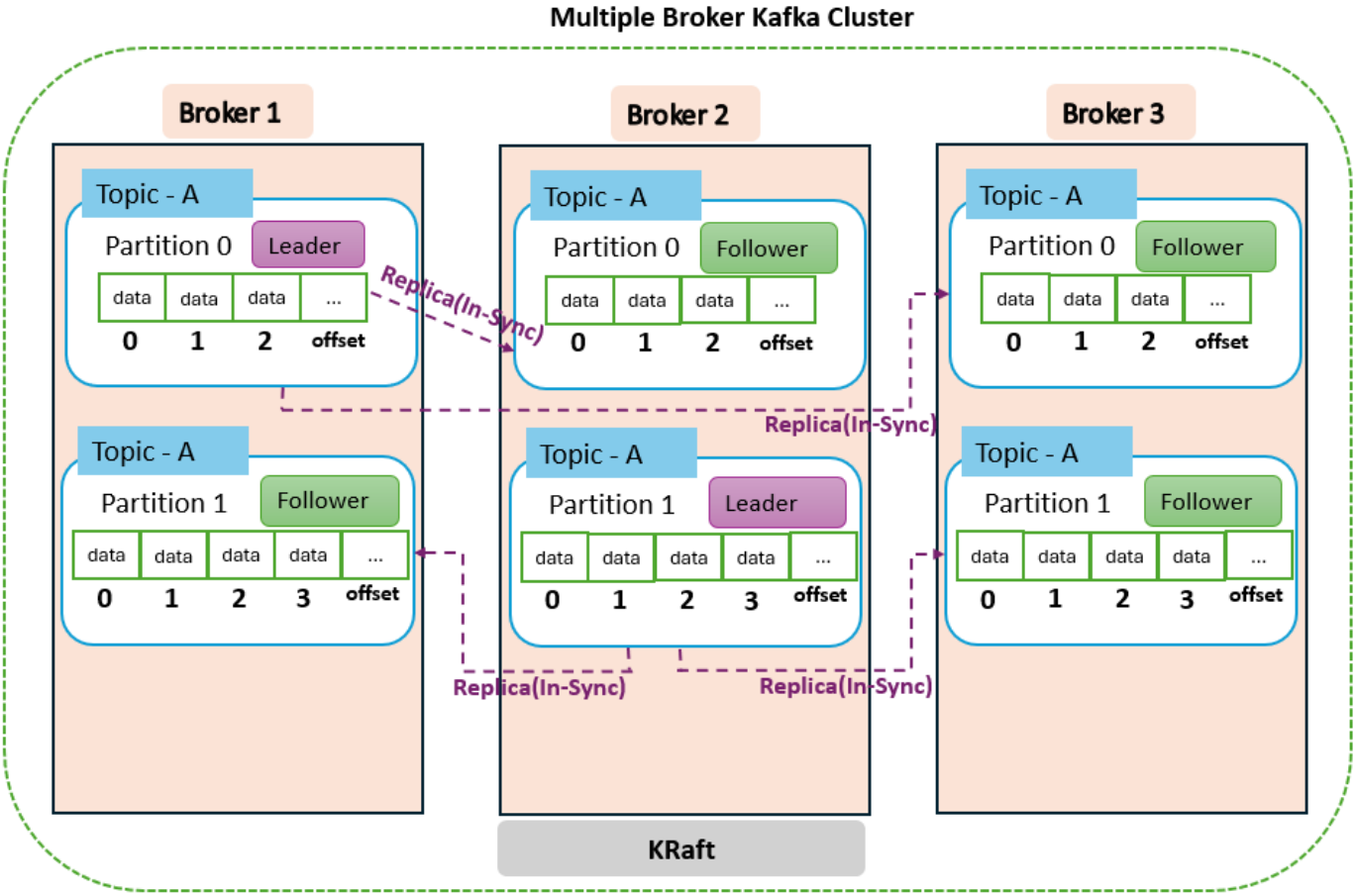
Single Broker with Replication Factor

In a Kafka setup with a single broker, the replication factor must be set to 1. This means each partition in Kafka has single leader and zero followers. The replication factor includes the total number of replicas including the leader. There is only one copy of each partition, and no replication occurs. This setup poses a significant risk of data loss and is not advisable for production environments.



Multiple Brokers with Replication Factor

In a multi-broker Kafka cluster, the replication factor can be set to a value greater than 1. This means that each partition will have multiple copies distributed across different brokers. It ensures fault tolerance, high availability and data durability.



Replication Factor 3 with three brokers and two partitions

Let's consider a Kafka cluster with 3 brokers and a topic A with 2 partitions (0 and 1). The replication factor is set to 3, meaning each partition will have 3 replicas (including leader).

Kafka Setup

- **Brokers**: Kafka cluster consists of 3 brokers, named Broker 1, Broker 2, and Broker 3.
- **Replication Factor**: Set to 3, meaning each partition will have 3 replicas (including leader).
- **Topic**: topic A with 2 partitions, named Partition 0 and Partition 1.

Partition and Replica Distribution

- Broker1 contains Leader for Partition 0, Follower for Partition 1
- Broker2 contains Follower for Partition 0, Leader for Partition 1

- Broker3 contains Follower for both Partition 0 and Partition 1

Data Flow

When a producer sends data to Topic A with Partition 0, it is first stored into the leader of Partition 0 on Broker 1. The data is then replicated (In-Sync Replica) to the follower partitions on both Broker 2 and Broker 3.

If Broker 1 fails, Kafka will select a new leader for Partition 0 from the in-sync replicas (either Broker 2 or Broker 3). This ensures that the system continues to function properly, maintaining data availability and durability.

Best practices for Kafka replication

- Starting with a replication factor of three and three brokers in a Kafka cluster is recommended to ensure even data distribution, fault tolerance and can survive the failure of up to two brokers.
- A good rule of thumb is to have at least as many brokers as the replication factor to ensure even distribution and fault tolerance
- Avoid setting up too high replication factor which lead to increased resource consumption and network traffic
- Avoid setting up too low replication factor which can compromise data availability and fault tolerance

4. Kafka Producer, Acknowledgements and Idempotency

Producers are clients that publish messages to Kafka topics, distributing them across various partitions. They send data to the broker, which then stores it in the corresponding partition of the topic.

Each message or record that a producer sends includes a Key (optional), Value, Header (optional), and Timestamp.

Message/Record Key

Message keys in Kafka are optional but can be quite beneficial. When a key is provided, Kafka hashes the key to determine the partition. This guarantees that all messages with the same key are directed to the same partition, which is crucial for maintaining order in message processing.

If a Kafka producer does not provide a message key, Kafka distributes the messages evenly across the available partitions using a round-robin algorithm. However, this approach helps balance the load across partitions but does not guarantee order for related messages.

Acknowledgments (acks)

Acknowledgements (acks) in Kafka are a mechanism to ensure that messages are reliably stored in the Kafka topic partition.

Kafka producers only write data to the leader partition in the broker.

Kafka producers must also set the acknowledgment level (acks) to indicate whether a message needs to be written to a minimum number of replicas before it is considered successfully written.

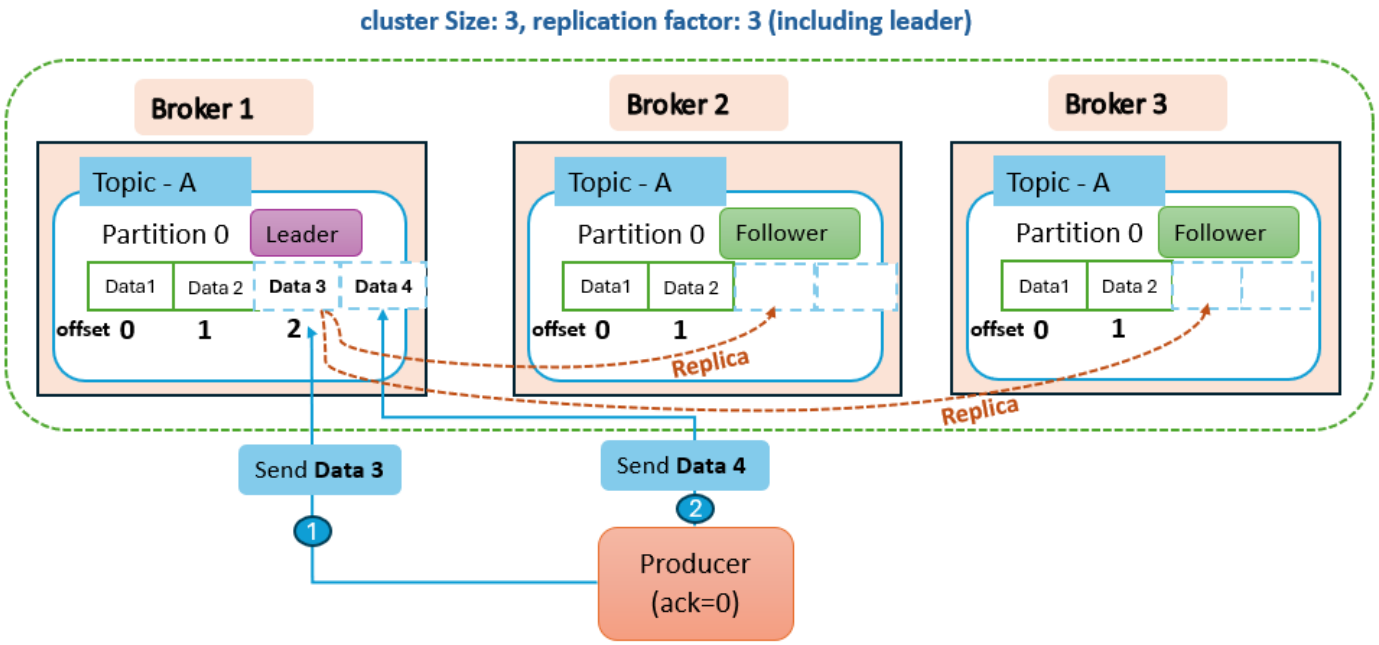
acks=0

The producer sends messages without waiting for any acknowledgment from the broker. While this approach minimizes latency, it also carries a high risk of message loss since the producer doesn't receive confirmation that the message was successfully written.

Message Delivery Semantics

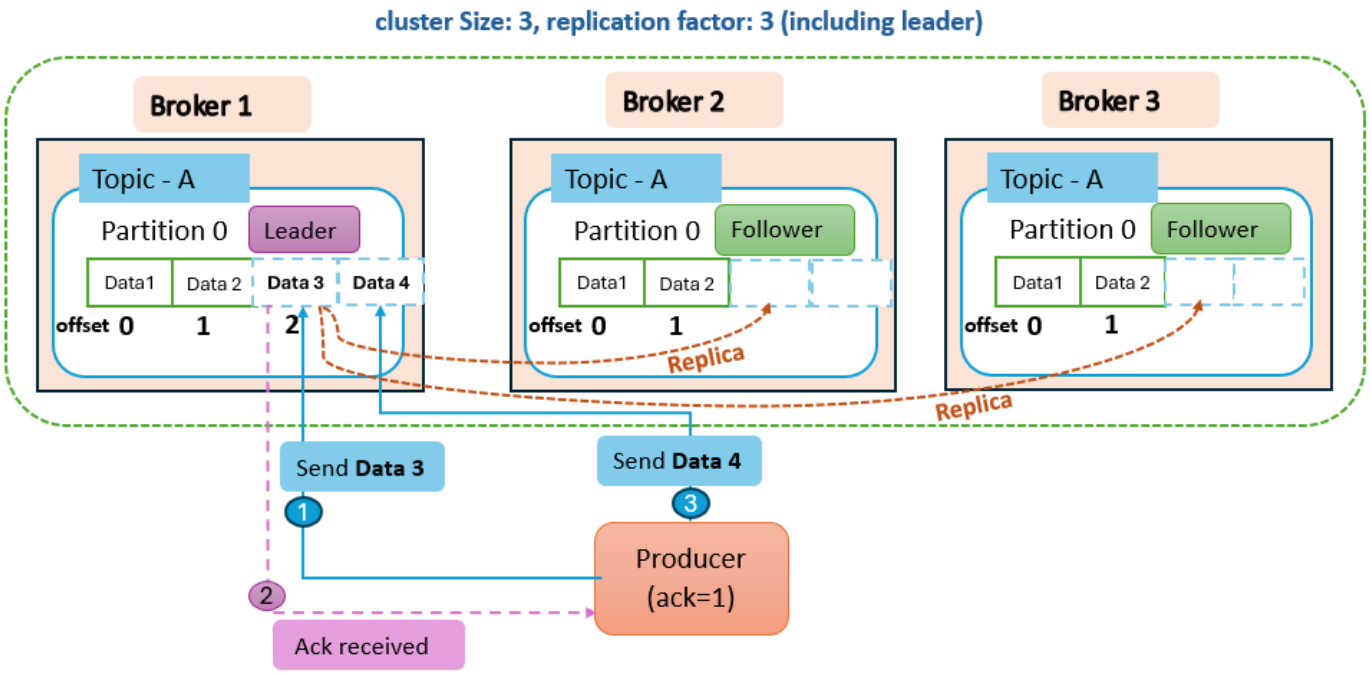
This follows the **At Most Once** delivery model, where messages are delivered once, and if a failure occurs, they may be lost and not redelivered. This approach is suitable for scenarios where occasional data loss is acceptable and low latency is crucial.

```
acks=0 // configuration
```



acks=1

The producer waits for an acknowledgment from the leader partition only before responding to the client request. This setting provides a balance between latency and reliability. However, the producer does not wait for the all in-sync replicas to be updated with latest data. Therefore, there is a risk of data loss if the leader partition fails before the in-sync replicas are updated.



acks=all

The producer waits for an acknowledgment from the leader Partition and all in-sync partition replicas before responding to the client request. This setting provides highest level of reliability and ensures that the message is replicated across multiple brokers. However, it can increase latency because the producer waits for acknowledgments from multiple brokers.

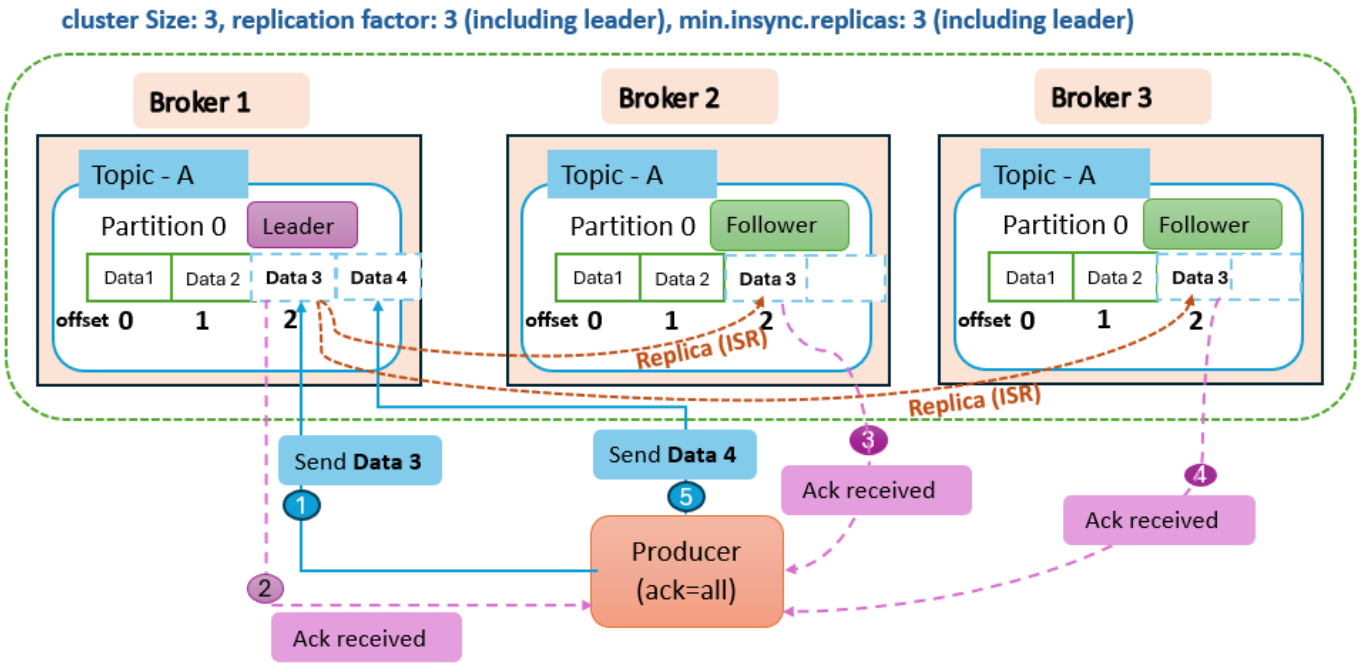
Message Delivery Semantics

This follows the **At Least Once** delivery model, where messages are delivered one or more times, and if a failure occurs, messages are not lost but may be delivered more than once. This approach is suitable for scenarios where data loss is unacceptable, and duplicates can be handled.

```
acks=all, retries=Integer.MAX_VALUE // configuration
```

Example #1:

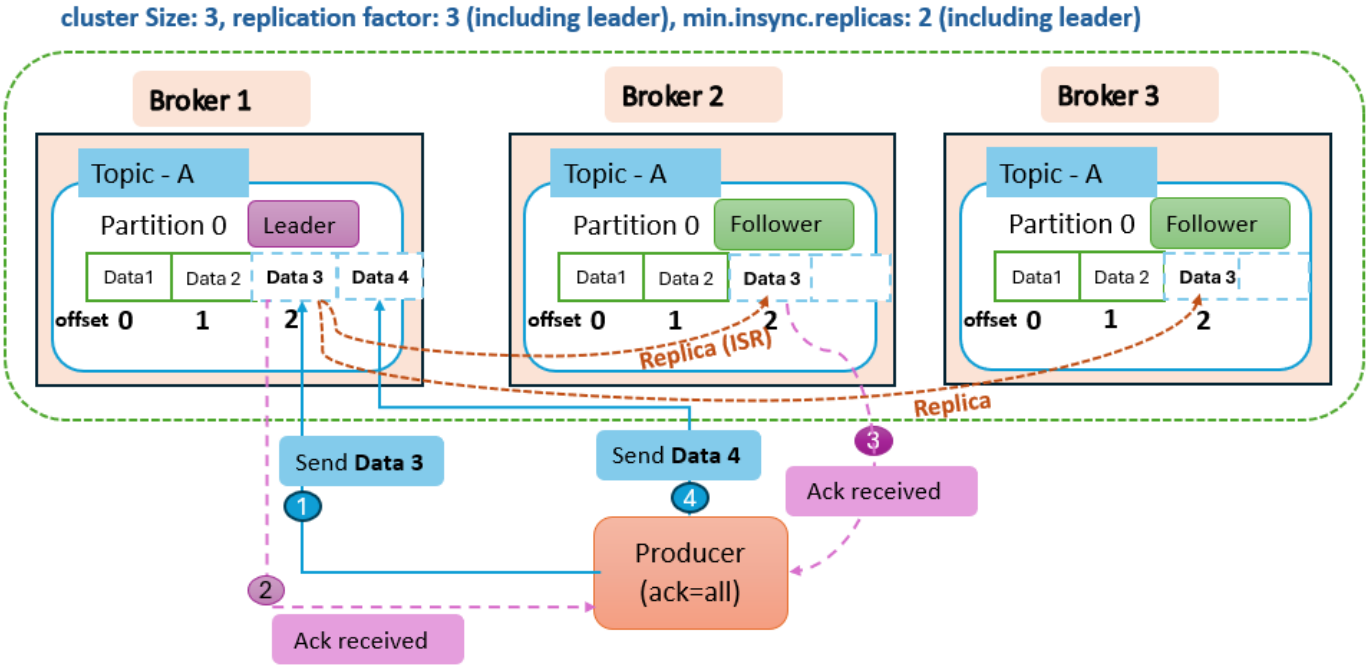
- cluster size: 3
- replication factor: 3 (including leader)
- min.insync.replicas: 3 (including leader)



When a producer sends a message to a partition, the leader broker writes the message and waits for acknowledgments from at least two follower replicas (resulting in a total of three acknowledgments: one from the leader and two from followers). Once this condition is met, the message is considered successfully written.

Example #2:

- cluster size: 3
- replication factor: 3 (including leader)
- min.insync.replicas: 2 (including leader)



When a producer sends a message to a partition, the leader broker writes the message and waits for acknowledgments from at least one follower replicas (resulting in a total of two acknowledgments: one from the leader and one from follower). Once this condition is met, the message is considered successfully written.

The widely used configuration is acks=**all** and min.insync.replicas=**2** for ensuring data durability and availability

Idempotency

Idempotency in Kafka producers guarantees that each message is delivered exactly once. This prevents duplicate entries during retries and maintains message ordering.

Without Idempotency

Imagine a scenario where a producer sends a message, but due to a network issue, the acknowledgment from the broker is not received. The producer will retry sending the message. This leads to duplicate message commits.

With Idempotency

Imagine a scenario where a producer sends a message, but due to a network issue, the acknowledgment from the broker is not received. The producer will retry sending the message. Here, the broker will identify the duplicate and discard it, ensuring that only one copy of the message is stored.

Kafka performs the following steps internally to ensure idempotency

- When idempotency is enabled, each producer is assigned a unique Producer ID (PID)
- Each message sent by the producer is assigned a monotonically increasing sequence number that is unique to each partition.
- The broker tracks the highest sequence number it has received from each producer for each partition. If it receives a message with a lower sequence number, it discards it as a duplicate.

Message Delivery Semantics

This follows the **Exactly Once** delivery model, where messages are exactly once, even in the case of retries. This ensures no duplicates and maintains message order. This approach is suitable for scenarios application requires strict data consistency and no duplicates.

```
acks=all, enable.idempotence=true // configuration
```

5. Kafka Consumer and Consumer Group

Consumer

Consumers are clients that read data from Kafka topics. They subscribe to one or more topics and process the data. Each consumer keeps track of its position in each partition using partition offset.

Kafka consumers follow a **pull model**. This means that consumers actively request data from Kafka brokers rather than having the brokers push data to them

Advantages of the Pull Model

- Consumers can control the rate at which they consume messages, allowing them to manage varying loads more effectively.

- Consumers have the capability to retrieve messages in batches, enhancing both network and processing efficiency.
- If a consumer runs into an error, it can attempt to fetch the messages again without impacting other consumers.

Consumer Group

A group of consumers work together to consume data from a topic. Each consumer in the group processes data from different partitions, allowing for parallel processing and load balancing.

Kafka consumers are typically part of a consumer group. When multiple consumers are subscribed to a topic and are part of the same group, each consumer will receive messages from a different subset of the partitions in the topic.

Here are some key points about consumer groups:

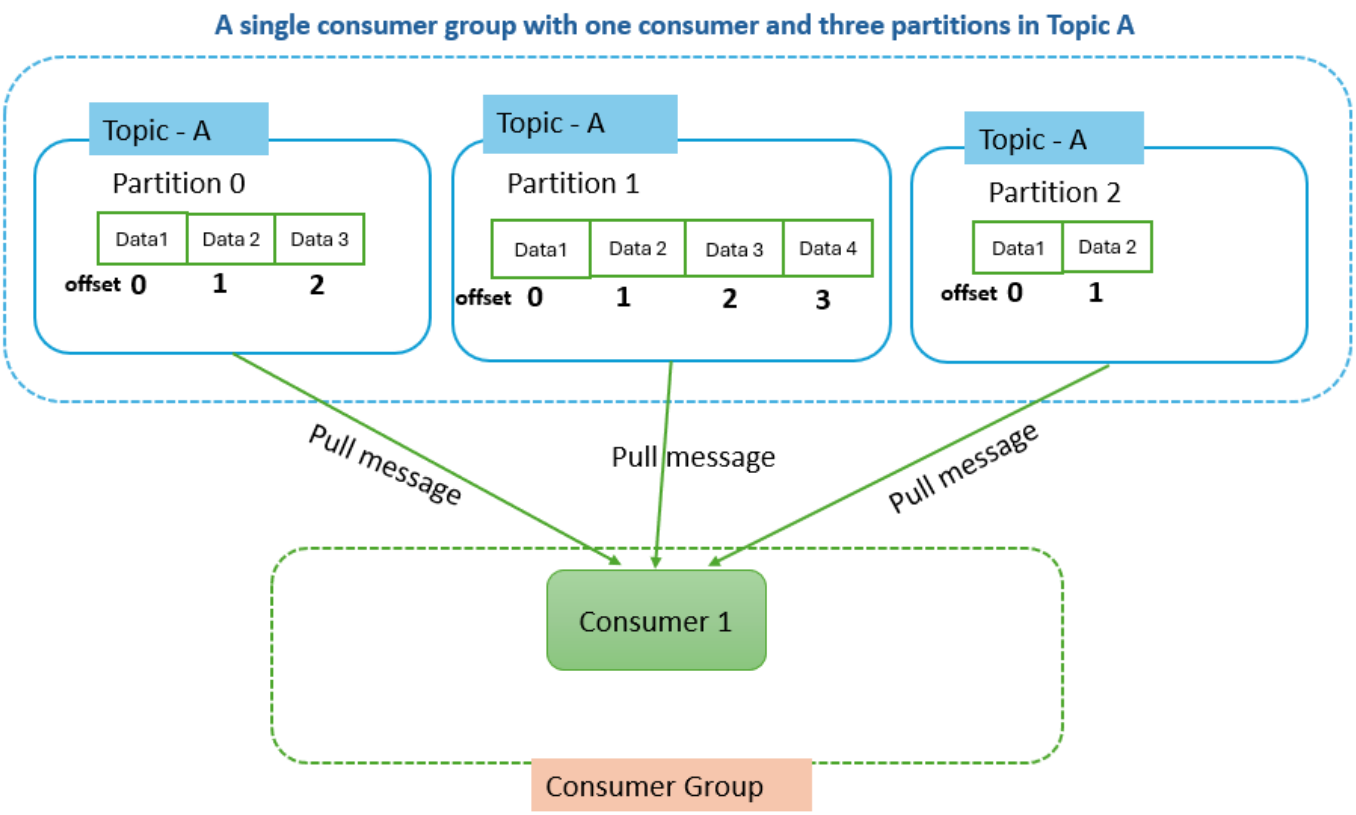
- Each partition of a topic is allocated to a single consumer within a group at any given time. This guarantees that each message is handled by only one consumer in the group.
- Consumer groups enable horizontal scaling. By adding more consumers to a group, Kafka will redistribute the partitions among the consumers, leading to more efficient data processing.
- If a consumer in a group fails, Kafka will automatically rebalance the partitions assigned to that consumer among the remaining consumers in the group.

Offset Management

The offsets for each partition are stored in the __consumer_offsets topic on the Kafka broker. This allows Consumer to resume reading from the correct position in case of a restart or failure.

Example #1: A single consumer group with one consumer and three partitions in Topic A

Consider a Kafka topic, Topic A, which is divided into three partitions: Partition **0**, Partition **1**, and Partition **2**. You also have a consumer group, Consumer **Group 1**, with only one consumer, **Consumer 1**.



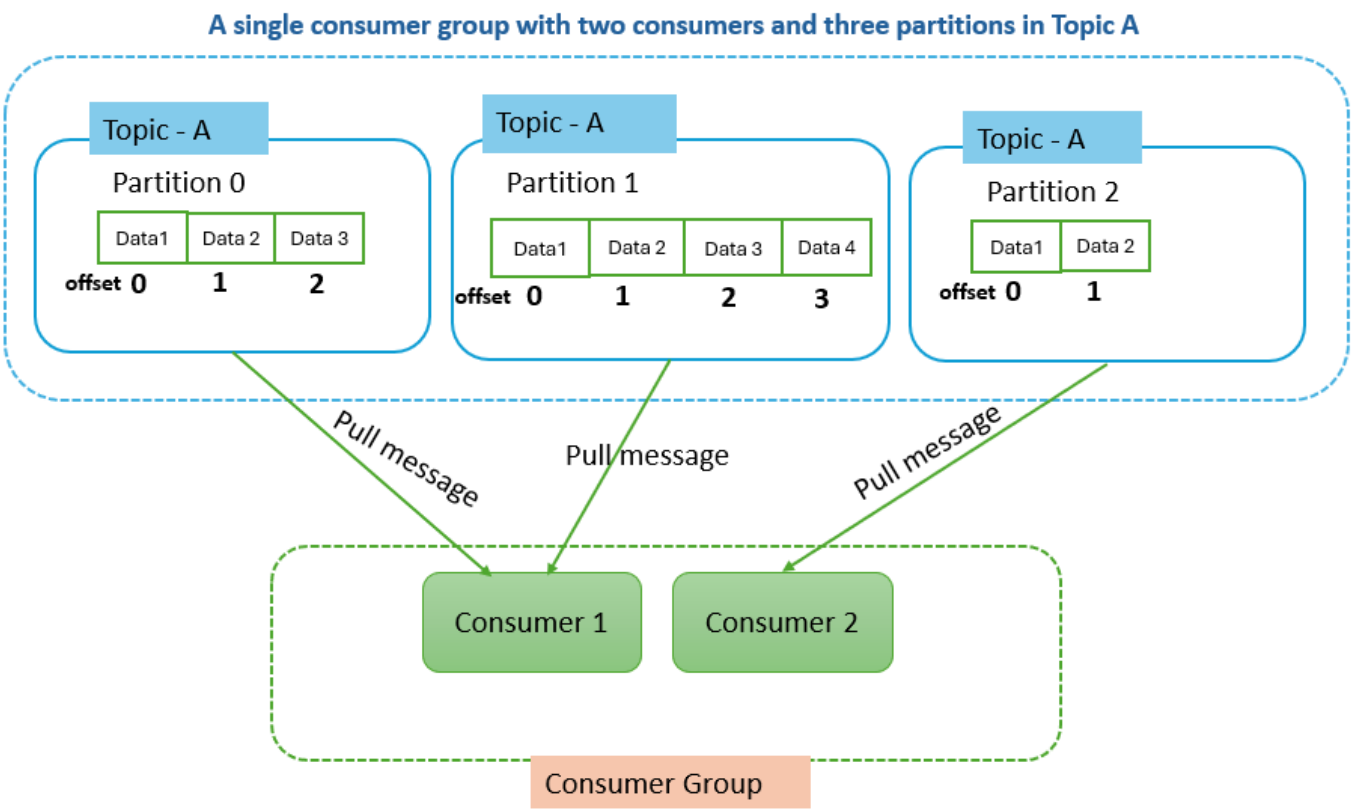
Since there is only one consumer in the group, Consumer 1 will be assigned all three partitions of Topic A. This means Consumer 1 will read messages from Partition 0, Partition 1, and Partition 2.

Disadvantage

- With just one consumer, there is no parallel processing of partitions. The single consumer handles all partitions, which could become a bottleneck if the message volume is high.

Example #2: A single consumer group with two consumers and three partitions in Topic A

Consider a Kafka topic, Topic A, which is divided into three partitions: Partition **0**, Partition **1**, and Partition **2**. You also have a consumer group, Consumer **Group 1**, with two consumers, **Consumer 1** and **Consumer 2**.



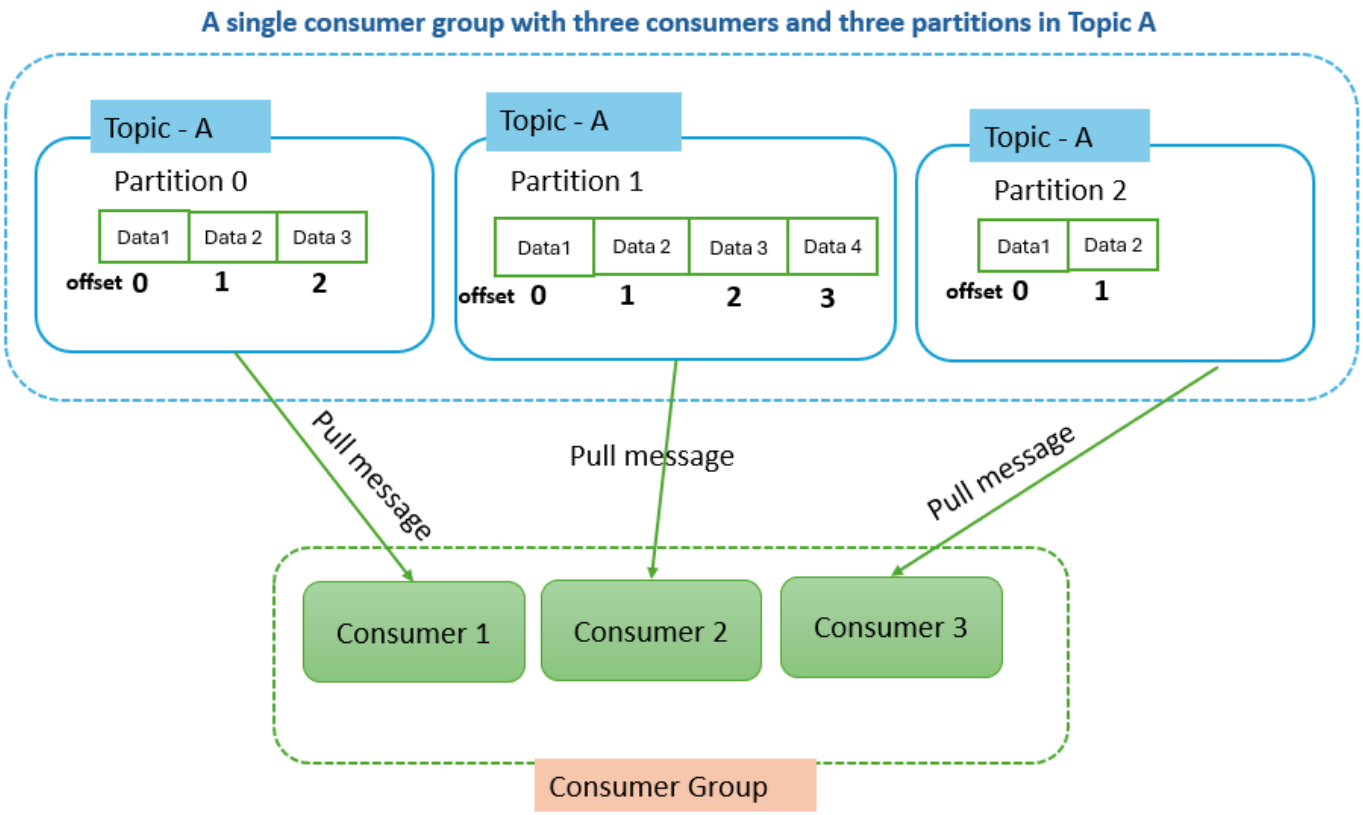
Since there are three partitions and two consumers, one consumer will be assigned two partitions, and the other consumer will be assigned one partition. It means **Consumer 1** might be assigned Partition **0** and Partition **1**, while **Consumer 2** is assigned Partition **2**.

Advantages

- With two consumers, the partitions are processed in parallel, improving the overall throughput and efficiency of message consumption
- The workload is shared among the consumers. However, because there is an odd number of partitions, one consumer will end up managing more partitions than the other.

Example #3: A single consumer group with three consumers and three partitions in Topic A

Consider a Kafka topic, Topic A, which is divided into three partitions: Partition **0**, Partition **1**, and Partition **2**. You also have a consumer group, Consumer **Group 1**, with three consumers, **Consumer 1**, **Consumer 2** and **Consumer 3**.



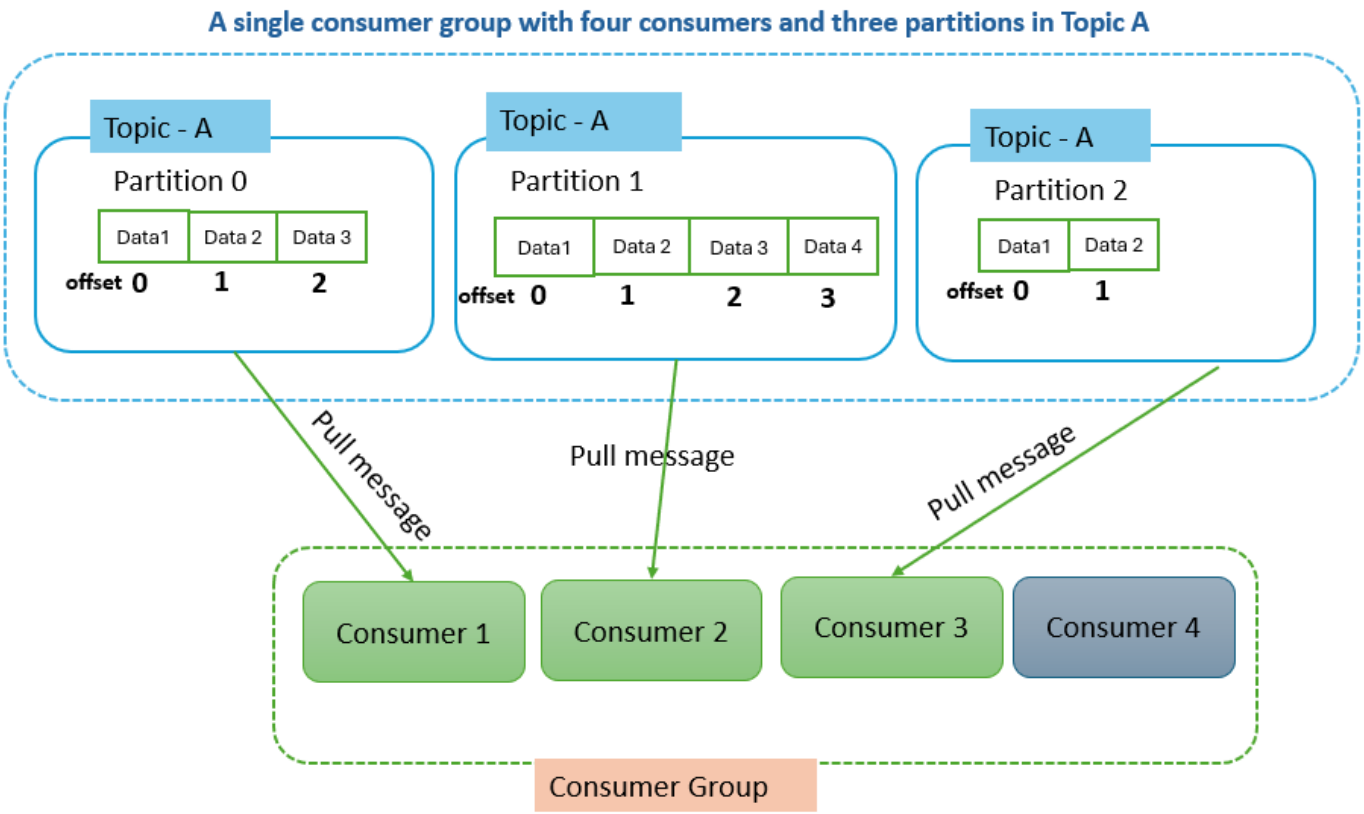
Since there are three partitions and three consumers, each consumer will be assigned exactly one partition. It means Consumer 1 might be assigned Partition 0, Consumer 2 might be assigned Partition 1 and Consumer 3 might be assigned Partition 2

Advantages

- With three consumers, the partitions are processed in parallel, maximizing the throughput and efficiency of message consumption.
- The load is evenly distributed among the consumers, with each consumer handling one partition.

Example #4: A single consumer group with four consumers and three partitions in Topic A

Consider a Kafka topic, Topic A, which is divided into three partitions: Partition **0**, Partition **1**, and Partition **2.** You also have a consumer group, Consumer **Group 1**, with four consumers, **Consumer 1**, **Consumer 2**, **Consumer 3** and **Consumer 4**.



Since there are three partitions and four consumers, one consumer will not be assigned any partition. It means Consumer 1 might be assigned Partition 0, Consumer 2 might be assigned Partition 1, Consumer 3 might be assigned Partition 2 and Consumer 4 will not be assigned any partition and will remain idle until a rebalance occurs.

Advantages

- With three consumers, the partitions are processed in parallel, maximizing the throughput and efficiency of message consumption.
- The load is evenly distributed among the consumers, with each consumer handling one partition.

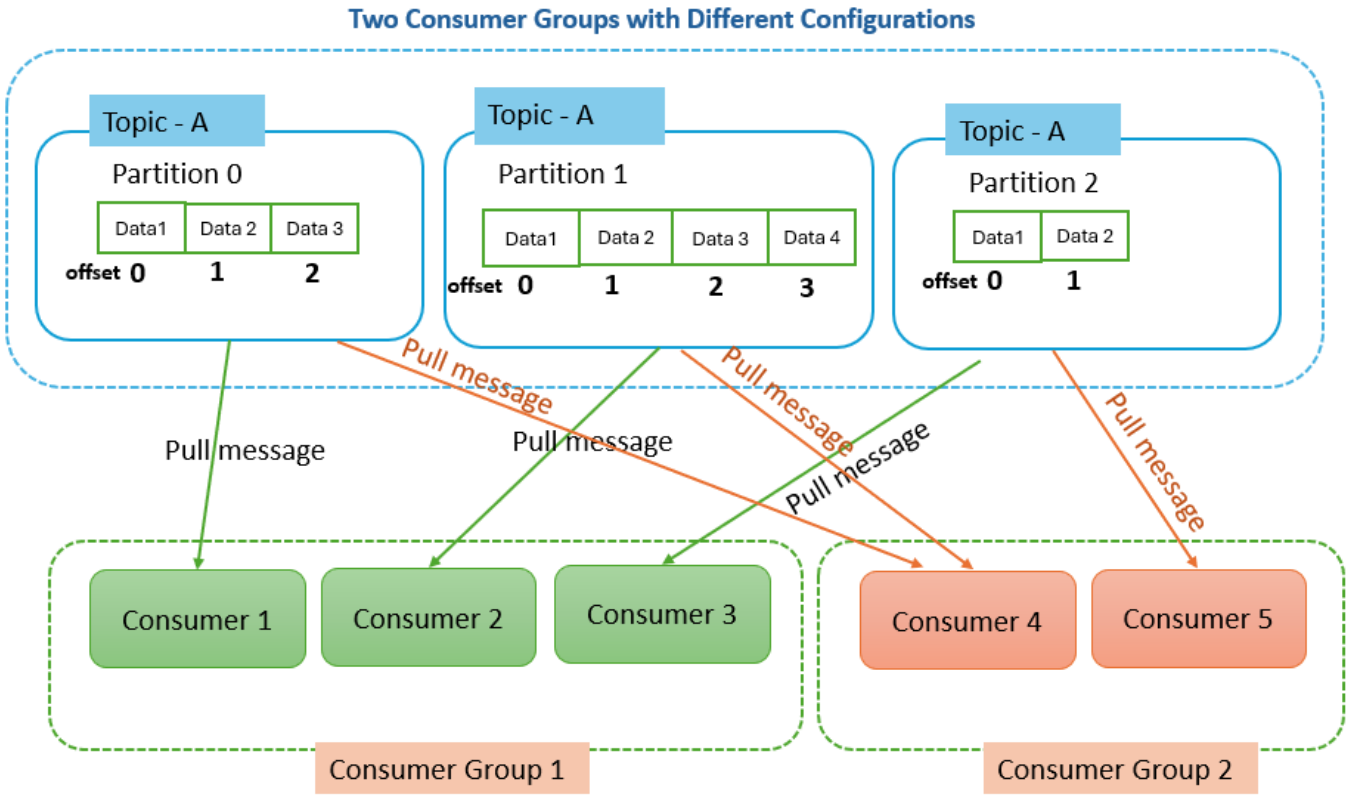
Disadvantage

- One consumer will remain idle as there are more consumers than partitions. This consumer will be available to take over if another consumer fails or if more partitions are added.

Example #5: Two Consumer Groups with Different Configurations

When you have multiple applications that need to process the same data differently, you can create separate consumer groups for each application. Each group will independently consume the messages and apply its own processing logic.

If one consumer group fails or lags behind, it does not affect the other groups



Message Delivery Semantics

Kafka provides three main message delivery semantics to ensure different levels of reliability and performance

- **At Most Once**: Messages are delivered once, and if there is a failure, they may be lost and not redelivered. This approach is suitable for scenarios where occasional data loss is acceptable and low latency is crucial.

```
enable.auto.commit=true  // configuration
```

- **At Least Once**: Messages are delivered one or more times, and if a failure occurs, messages are not lost but may be delivered more than once. This approach is suitable for scenarios where data loss is unacceptable, and duplicates can be handled.

```
enable.auto.commit=false  // configuration
Consumer should commit offsets after processing messages to ensure they are not lost.
```

- **Exactly Once**: Messages are delivered exactly once, even in the case of retries. This ensures no duplicates and maintains message order. This approach is suitable for scenarios application requires strict data consistency and no duplicates.

```
isolation.level=read_committed // configuration
```

6. Kafka Consumer Partition Rebalancing

Partition rebalancing in Kafka is essential for maintaining an even distribution of data across consumers within a consumer group. Partition rebalancing can occur in the following situations

New Consumer Joins the Group

When a new consumer joins the group, Kafka needs to redistribute the partitions to include the new consumer.

Consumer Leaves the Group

If a consumer crashes or is shut down, Kafka redistributes its partitions among the remaining consumers.

Partition Changes in a Topic

Adding or removing partitions in a topic also triggers a rebalance

Kafka Rebalancing Protocols

Eager Rebalancing Protocol:

It is also known as "stop-the-world" rebalancing, this protocol stops all consumers in the group

during a rebalance. All consumers must rejoin the group and receive new partition assignments.

It is simple and straightforward. However, it can cause significant disruption and downtime, especially in large clusters, as all consumers must stop and rejoin the group

Incremental Cooperative Rebalancing Protocol

This protocol supports incremental rebalancing, so only the consumers that need to adjust their assignments will pause and rejoin the group, while the rest continue processing.

It reduces disruption and downtime, making it ideal for large clusters. However, it is more complex to implement and manage compared to eager rebalancing.

Partition assignment strategies

During a rebalance, Kafka uses partition assignment strategies to determine how partitions are assigned to consumers. Here are the different rebalance strategies

Round-Robin Assignment

This is the simplest and most commonly used partition assignment strategy in Kafka. Partitions are assigned to consumers one after another in a circular fashion. This approach helps balance the load evenly among consumers, ensuring that no single consumer is overloaded while others remain underutilized.

This assignment uses **Eager Rebalancing Protocol** while rebalancing.

Use below setting to enable round-robin assignment

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor
```

Example:

Consider a scenario with 3 consumers and 6 partitions:

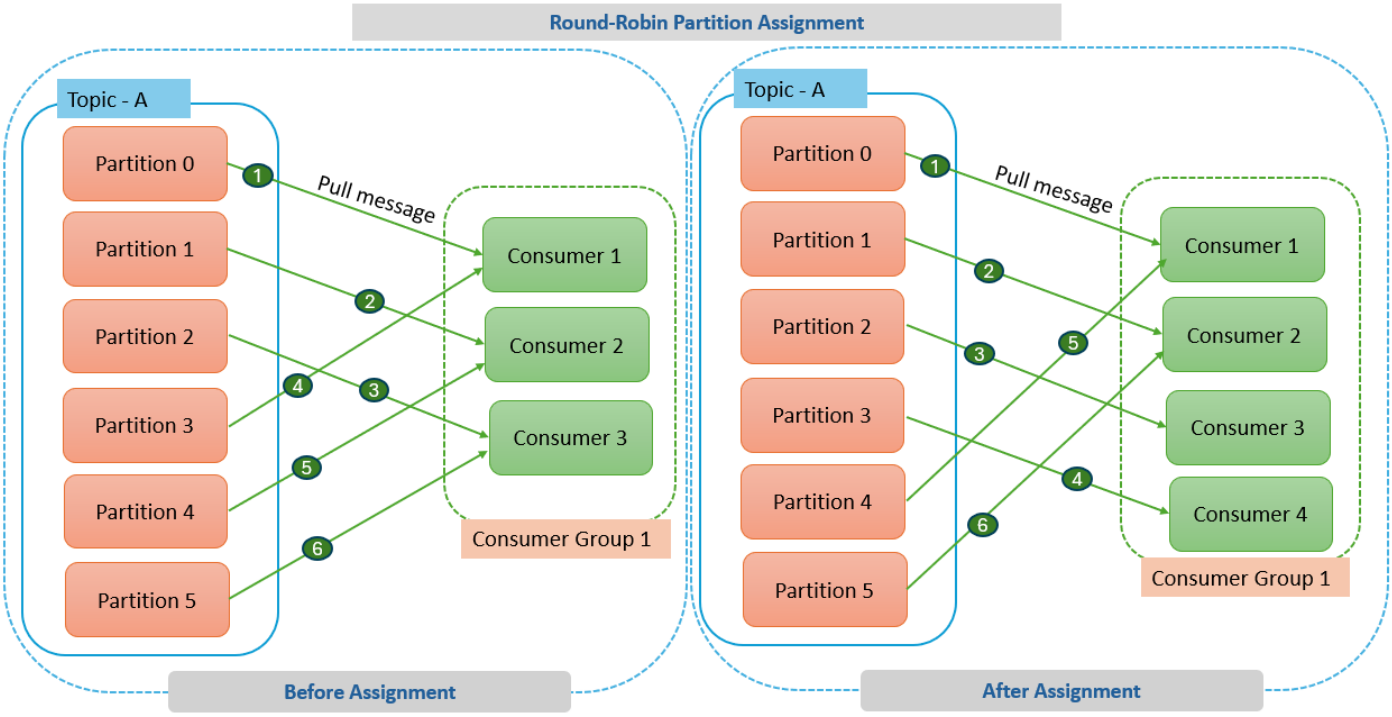
- Topic: A
- Consumers: Consumer 1, Consumer 2, Consumer 3
- Partitions: Partition 0, Partition 1, Partition 2, Partition 3, Partition 4, Partition 5

Assignment before a new consumer joins:

Consumer 1: Partition 0, Partition 3
Consumer 2: Partition 1, Partition 4
Consumer 3: Partition 2, Partition 5

Assignment after a new consumer C4 joins:

Consumer 1: Partition 0, Partition 4
Consumer 2: Partition 1, Partition 5
Consumer 3: Partition 2
Consumer 4: Partition 3



This assignment is suitable for evenly distributed partitions and consumers with similar processing capabilities

Range Assignment

This approach allocates partitions to consumers in contiguous ranges, making it beneficial when partitions have a natural order and consumers can efficiently manage a range of partitions. Partitions are sorted and divided into contiguous ranges, with each range assigned to a specific consumer.

This assignment uses **Eager Rebalancing Protocol** while rebalancing.

Use below setting to enable range assignment

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor
```

Example:

Consider a scenario with 3 consumers and 6 partitions:

- Topic: A
- Consumers: Consumer 1, Consumer 2, Consumer 3
- Partitions: Partition 0, Partition 1, Partition 2, Partition 3, Partition 4, Partition 5

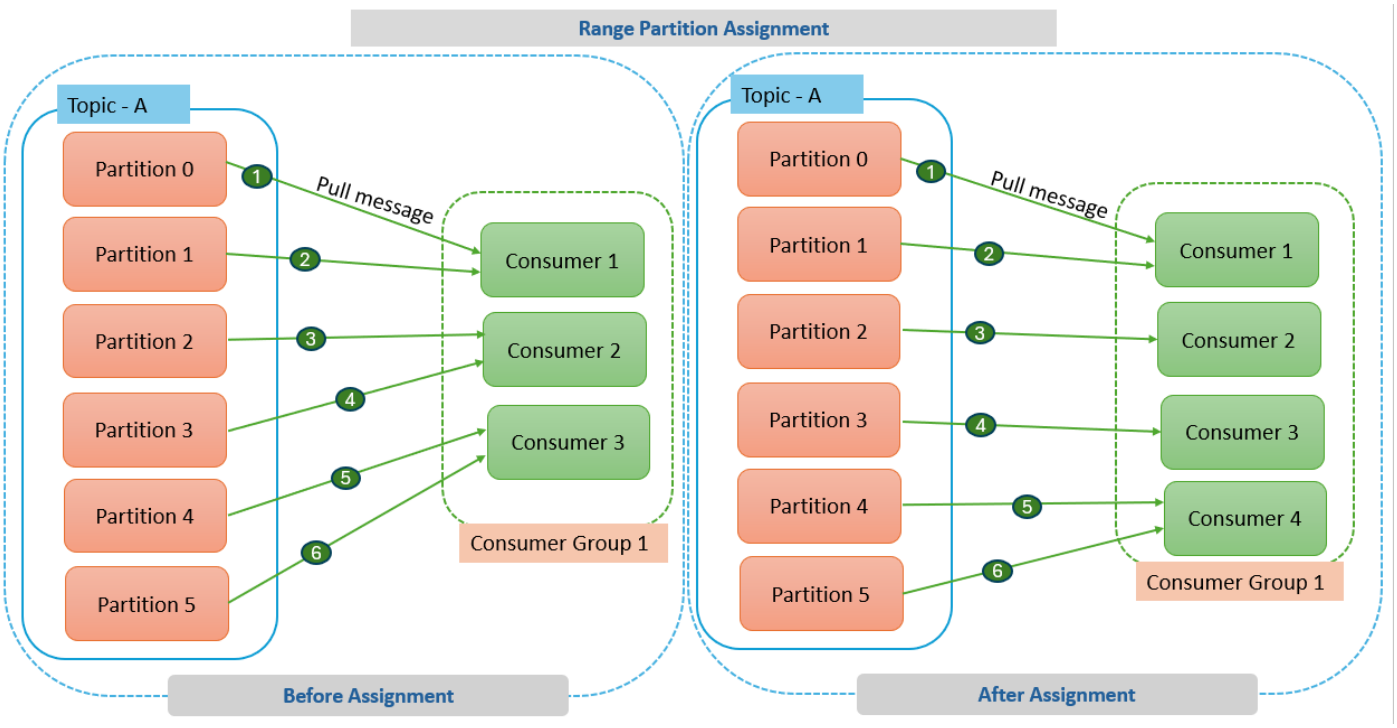
Assignment before a new consumer joins:

Consumer 1: Partition 0, Partition 1
Consumer 2: Partition 2, Partition 3
Consumer 3: Partition 4, Partition 5

Each consumer gets a contiguous range of partitions

Assignment after a new consumer C4 joins:

Consumer 1: Partition 0, Partition 1
Consumer 2: Partition 2
Consumer 3: Partition 3
Consumer 4: Partition 4, Partition 5



This assignment is suitable when partitions have a natural order, and consumers can handle a range of partitions efficiently.

Sticky Assignment

The approach aims to minimize partition movement by sticking to previous assignments as much as possible. It reduces the overhead of rebalancing and helps maintain data locality.

This assignment uses **Eager Rebalancing Protocol** while rebalancing.

Use below setting to enable sticky assignment

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.StickyAssignor
```

Example:

Consider a scenario with 3 consumers and 6 partitions:

- Topic: A
- Consumers: Consumer 1, Consumer 2, Consumer 3
- Partitions: Partition 0, Partition 1, Partition 2, Partition 3, Partition 4, Partition 5

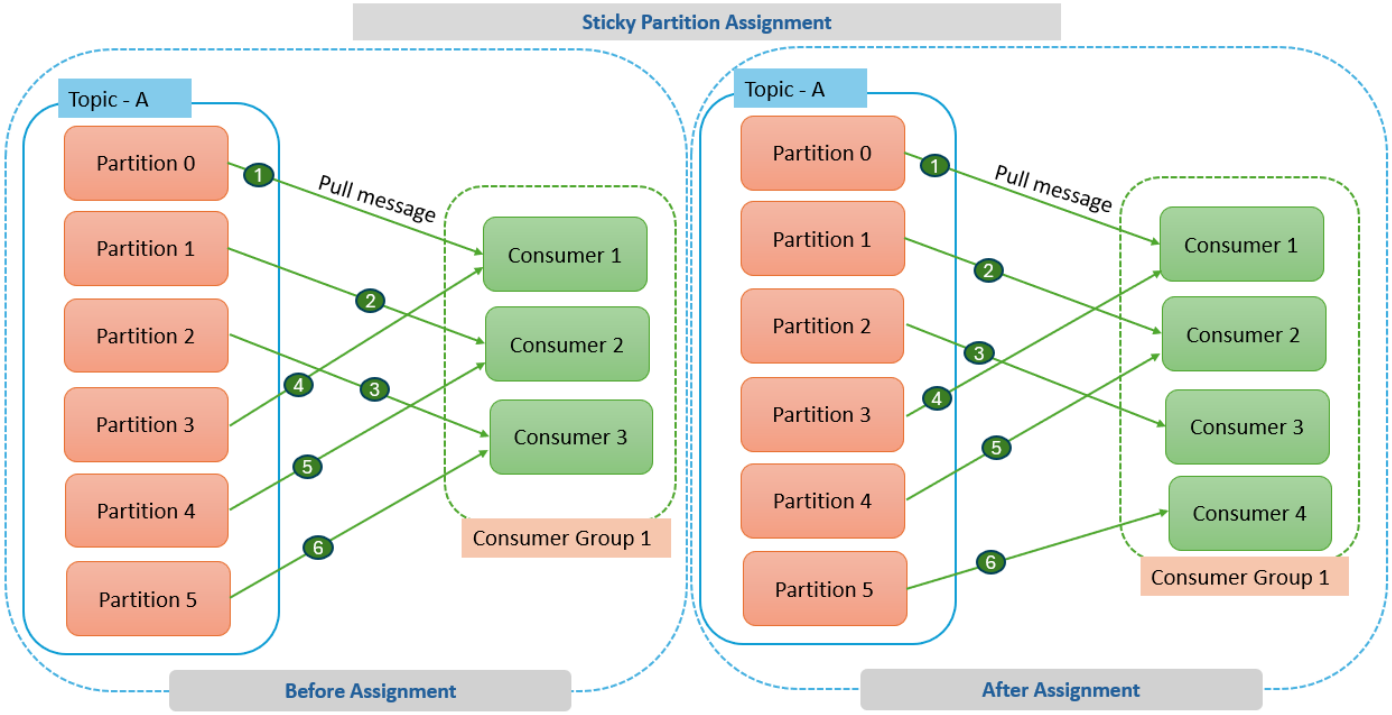
Assignment before a new consumer joins:

Consumer 1: Partition 0, Partition 3
Consumer 2: Partition 1, Partition 4
Consumer 3: Partition 2, Partition 5

Each consumer gets two partitions, distributed in a round-robin fashion initially.

Assignment after a new consumer C4 joins:

Consumer 1: Partition 0, Partition 3
Consumer 2: Partition 1, Partition 4
Consumer 3: Partition 2
Consumer 4: Partition 5



This assignment is suitable Ideal for reducing the overhead of rebalancing and maintaining data locality.

Cooperative Sticky Assignment

The approach aims to minimize partition movement by sticking to previous assignments as much as possible. It reduces the overhead of rebalancing and helps maintain data locality.

This assignment uses **Incremental Cooperative Rebalancing Protocol** while rebalancing.

Use below setting to enable cooperative sticky assignment

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
```

Example:

Consider a scenario with 3 consumers and 6 partitions:

- Topic: A
- Consumers: Consumer 1, Consumer 2, Consumer 3
- Partitions: Partition 0, Partition 1, Partition 2, Partition 3, Partition 4, Partition 5

Assignment before a new consumer joins:

Consumer 1: Partition 0, Partition 3
Consumer 2: Partition 1, Partition 4
Consumer 3: Partition 2, Partition 5

Each consumer gets two partitions, distributed in a round-robin fashion initially.

Assignment after a new consumer C4 joins:

Consumer 1: Partition 0, Partition 3
Consumer 2: Partition 1, Partition 4
Consumer 3: Partition 2
Consumer 4: Partition 5

The key difference is Sticky Assignment uses an **Eager Rebalancing Protocol**, while Cooperative Sticky Assignment uses **Incremental Cooperative Protocol**.

This assignment is suitable for large clusters where minimizing disruption during rebalancing is critical

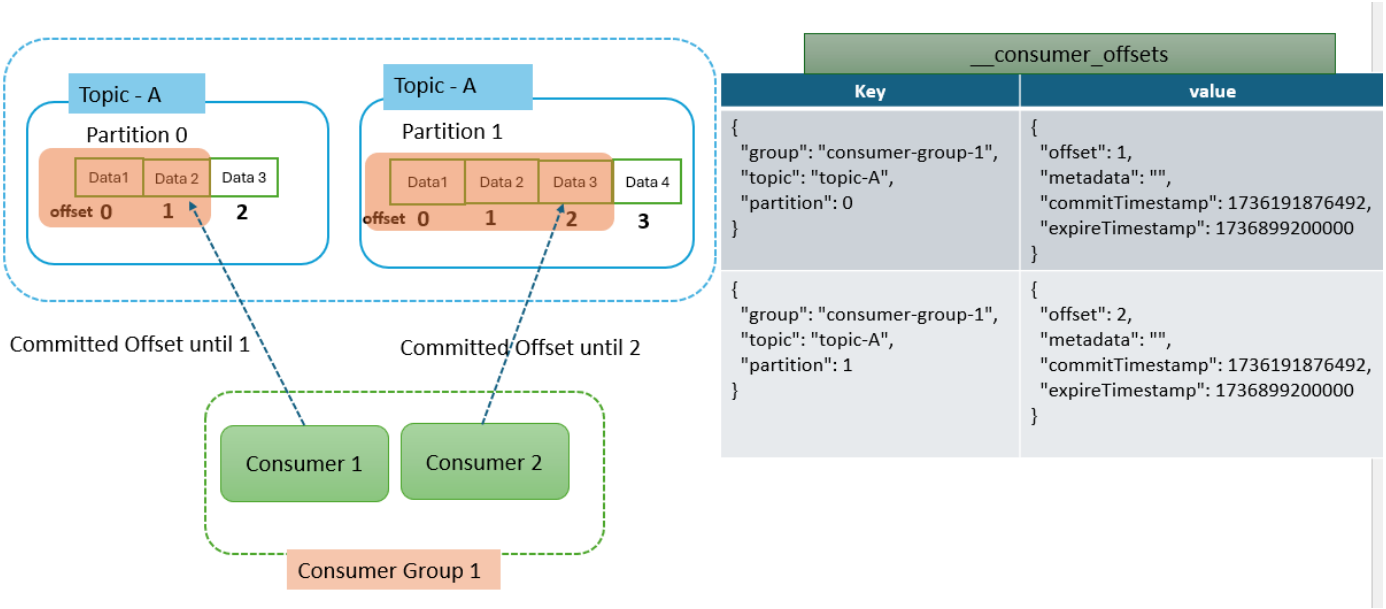
In Kafka, the default partition assignment strategy for consumers is the **Range Assignor**. This strategy allocates partitions to consumers in contiguous ranges, ensuring that each consumer receives a set of numerically adjacent partitions.

7. Kafka Consumer Commit Offset

Committing offsets is important to ensure that a consumer can restart from the correct position after a failure or restart. Without committing offsets, the consumer might reprocess messages, cause duplicates, or miss messages, leading to data loss.

Kafka brokers use an internal topic called **__consumer_offsets** to keep track of the last successfully processed messages for each consumer group. It contains only the most recent offset metadata for each consumer group, ensuring efficient storage and quick access.

The __consumer_offsets topic stores data as key-value pairs. The key includes details about the consumer group, topic, and partition, while the value contains the offset and metadata.



Kafka provides several ways to commit consumer offsets:

Auto Commit

Kafka automatically commits the offsets of messages consumed by a consumer at regular intervals. By default, this interval is set to 5000 milliseconds (5 seconds).

To enable auto commit, configure consumer with the following properties:

```
enable.auto.commit: true
auto.commit.interval.ms: 5000  // default 5000 milliseconds i.e. 5 seconds
```

It is simple and easy to implement. However, If the consumer crashes before the next auto commit interval, messages that were processed but not yet committed may be reprocessed again, leading to potential duplicates.

Example: Set up a Kafka consumer with auto commit enabled in Node.js using the kafkajs library.

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
```

```
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const run = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic
    await consumer.subscribe({ topic: 'topic-A', fromBeginning:
true });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                offset: message.offset, key:
message.key.toString(), value: message.value.toString()
            });
        },
    });
};

run().catch(console.error);
```

Manual Commit

It allows the consumer to manually commits offsets after processing messages. This provides greater control over when offsets are committed, reducing the risk of data loss or duplicate message processing.

To enable manual commit, configure consumer with the following properties:

```
enable.auto.commit: false
```

Manual Commit can be done either synchronously or asynchronously.

Synchronous Manual Commit

In this process, the consumer waits for the commit to complete before processing the next batch of messages, ensuring offsets are securely stored in Kafka before proceeding.

This can be implemented using commitSync() method in Java or waiting for the commitOffsets promise to resolve in kafkajs.

Example: Set up a Kafka consumer with synchronous manual commit in Node.js using the kafkajs library.

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: false
});

const run = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic
    await consumer.subscribe({ topic: 'topic-A', fromBeginning: true });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                offset: message.offset, key: message.key.toString(), value:
message.value.toString()
            });

            // Synchronous commit: Waits for the commit to complete
            await consumer.commitOffsets([
                { topic, partition, offset: (parseInt(message.offset, 10) +
1).toString() }
```

```
            ]);
        },
    });
};

run().catch(console.error);
```

Asynchronous Manual Commit

In this process, the consumer doesn't wait for the commit to finish to process the next batch of messages. This improves performance and throughput by reducing the latency associated with waiting for the commit operation to finish. However, it slightly increases the risk of data loss compared to synchronous commits.

This can be implemented using commitAsync() method in Java or do not await for the commitOffsets promise to resolve in kafkajs.

Example: Set up a Kafka consumer with asynchronous manual commit in Node.js using the kafkajs library.

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: false
});

const run = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic
    await consumer.subscribe({ topic: 'topic-A', fromBeginning: true });
```

```
    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                offset: message.offset, key: message.key.toString(), value:
message.value.toString()
            });

            // Asynchronous commit: Do not await for the commit to
complete
            consumer.commitOffsets([
                { topic, partition, offset: (parseInt(message.offset, 10) +
1).toString() }
            ]);
        },
    });
};

run().catch(console.error);
```

8. Kafka Consumer Auto Offset Reset

Consumer Auto Offset Reset configuration is mainly used in the following scenarios

- When a new consumer group is created and reads from the topic partition for the first time and consumer instances need to determine where to begin reading from the topic's partitions.
- When the last committed offset is deleted from the __**consumer_offsets** topic due to retention policy

When a consumer starts without a previously committed offset, the `**auto.offset.reset**` setting determines where it should begin reading messages. The possible values are:

earliest

In this setting, the consumer starts reading from the earliest available message in the partition. This is useful when you want to process all messages from the beginning.

This setting is perfect for applications that need to process all historical data, such as data analytics or batch processing jobs. However, It can cause high latency if there's a large number of messages, as the consumer will start from the very beginning.

use below settings to configure Auto Offset Reset

```
auto.offset.reset: earliest // in Java
fromBeginning: true   // in node.js using kafkajs library
```

Example: Set up a Kafka consumer with auto.offset.reset to **earliest** in Node.js using the kafkajs library

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const run = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic and set fromBeginning to true
    await consumer.subscribe({ topic: 'topic-A', fromBeginning: true });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                offset: message.offset, key: message.key.toString(), value:
message.value.toString()
            });
        },
    });
};

run().catch(console.error);
```

latest

In this setting, the consumer begins reading from the latest message upon subscribing to the topic partition. This default setting is ideal when you only need to process new messages.

Ideal for real-time applications like monitoring systems or real-time analytics, However, if the consumer is down while processing and committing the first message, it might miss messages produced during its downtime.

use below settings to configure Auto Offset Reset

```
auto.offset.reset: latest // in Java
fromBeginning: false // in node.js using kafkajs library
```

Example: Set up a Kafka consumer with auto.offset.reset to **latest** in Node.js using the kafkajs library

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const run = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic and set fromBeginning to false
    await consumer.subscribe({ topic: 'topic-A', fromBeginning: false });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                offset: message.offset, key: message.key.toString(), value:
message.value.toString()
            });
        },
```

```
    });
};

run().catch(console.error);
```

none

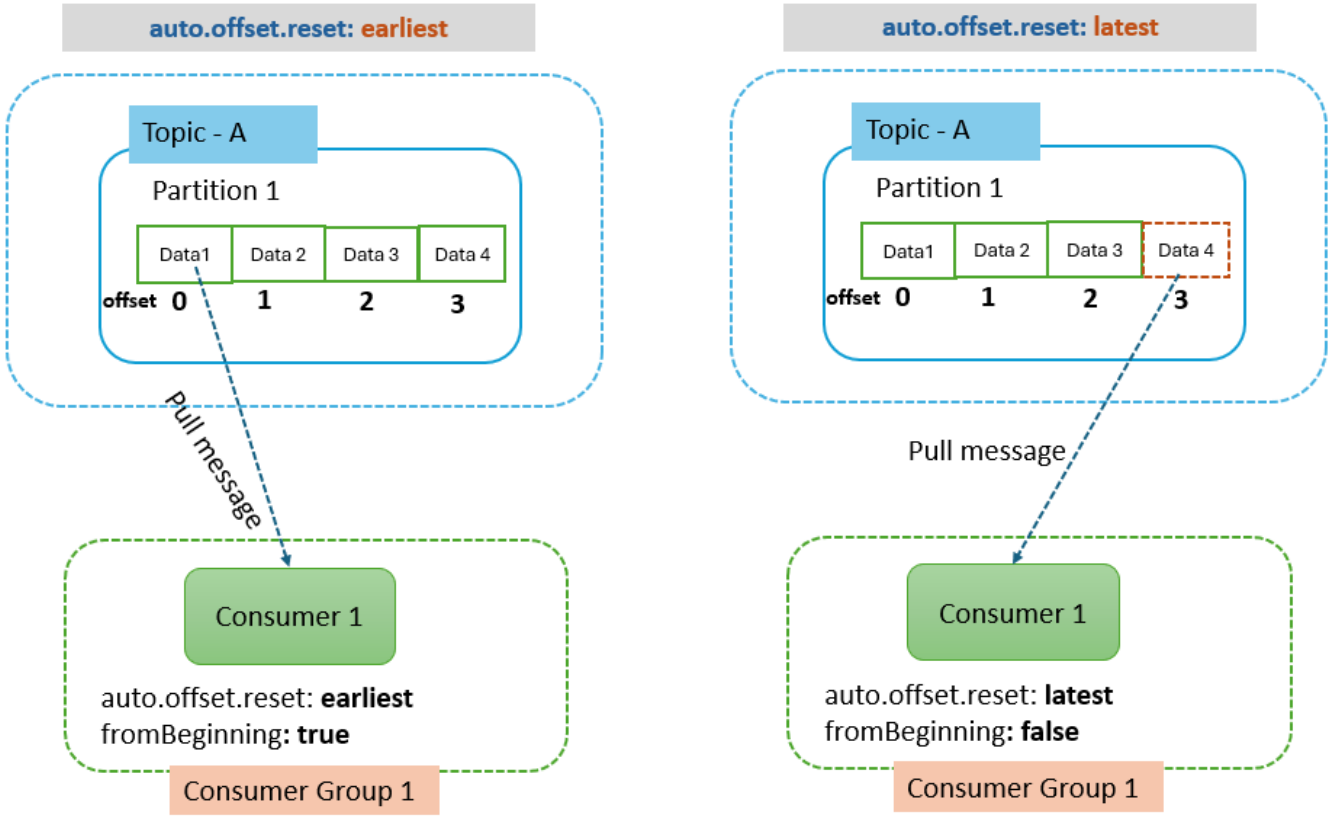
In this setting, the consumer throws an exception if no previous offset is found for the given consumer group. This is useful when you want to ensure that the consumer only processes messages if there is a previously committed offset.

Ideal for strict processing needs where the consumer should only start with a previously committed offset, However, it may lead to application errors if no offset is found.

Example: Set up a Kafka consumer with auto.offset.reset to **none** in Node.js using the kafkajs library

```
// install kafkajs using  'npm install kafkajs'

// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'test-app',
    brokers: ['localhost:9092']
});

const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const run = async () => {
    await consumer.connect();
    await consumer.subscribe({ topic: 'topic-A' });

    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {

            // throw error if no previous offset found
            if (message.offset === '0') {
                throw new Error('No previous offset found');
            }
            console.log({
```

```
                partition,
                offset: message.offset,
                value: message.value.toString(),
            });
        },
    });
};

run().catch(console.error);
```



Once a consumer reads a message and commits its offset, the "auto.offset.reset" setting is no longer relevant, as the committed offset now serves as the new starting point.

9. Replacing ZooKeeper with KRaft

In a traditional Kafka cluster setup, **ZooKeeper** is essential for managing and coordinating the Kafka Cluster. Its responsibilities include

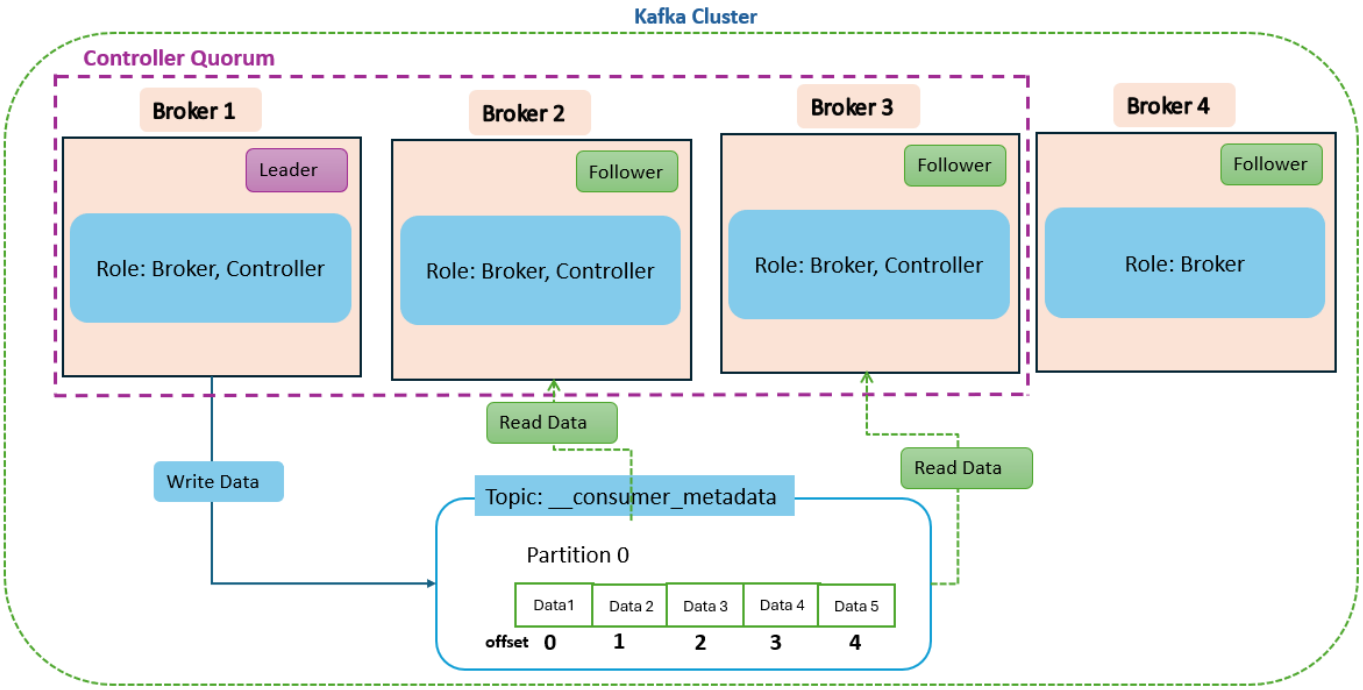
- Storing metadata about Kafka brokers, topics, partitions, and their configurations
- Maintaining Kafka topic information, such as the number of partitions, replication factor, and partition leader.
- Electing leaders for each partition to ensure there is always a leader available to handle read and write requests.
- Electing one of the nodes as the Kafka Controller, which manages the leader-follower relationship for partitions.
- Monitoring the health of Kafka brokers and notifying the controller of any broker failures, enabling quick failover and recovery. Maintaining Access Control Lists (ACLs) for all topics in the cluster.

However, there are several complexities to using ZooKeeper

- Kafka and ZooKeeper are separate systems, which adds complexity and increases the risk of misconfiguration when managing a Kafka cluster.
- Storing metadata in ZooKeeper can become a bottleneck as the Kafka cluster grows.
- Loading metadata from ZooKeeper can be slow, particularly during startup or controller elections.
- Synchronizing metadata between ZooKeeper and Kafka requires careful handling during version updates.

Introduction of KRaft(Kafka Raft)

Apache Kafka Raft(KRaft) is the consensus protocol introduced to remove Zookeeper for cluster metadata management. The Kafka KRaft architecture simplifies the metadata management within Kafka itself to remove the external system dependency.



Advantages of KRaft

- It uses a quorum-based controller, ensures that metadata is consistently replicated across the cluster
- The removal of ZooKeeper simplifies operational tasks, making it easier to monitor, administer, and troubleshoot Kafka clusters
- KRaft allows kafka to scale more efficiently even if cluster reaches millions of partitions
- It allows a single security model for the entire system
- It is production ready from Kafka version 3.3.1 onwards
- During startup or controller failover, a new controller can be spin up immediately because the data is already replicated across other controllers in the quorum

How KRaft Works

Quorum Controllers:

- Metadata is managed by a group of nodes known as quorum controllers.
- These controllers use the Raft consensus algorithm to ensure all nodes agree on the metadata state.

Event-Driven Protocol:

- Quorum controllers use an event-driven protocol to replicate metadata changes across all controllers in the quorum.

Metadata Storage:

- All metadata changes are stored in a dedicated Kafka topic called **__cluster_metadata**.
- This topic has a single partition containing all information related to topics, partitions, and configurations.

Leader-Follower Model:

- One of the quorum controllers acts as the leader and manages the metadata.
- The follower controllers replicate the metadata for failover purposes.

Commitment of Changes:

- For every metadata change, the leader controller sends an acknowledgment to all follower controllers.
- The change is considered committed only after receiving confirmation from the majority controllers.

Leader Election:

- If the leader becomes unreachable, the followers initiate a new leader election.
- A follower becomes a candidate, requests votes from other nodes, and the node with the majority of votes becomes the new leader.

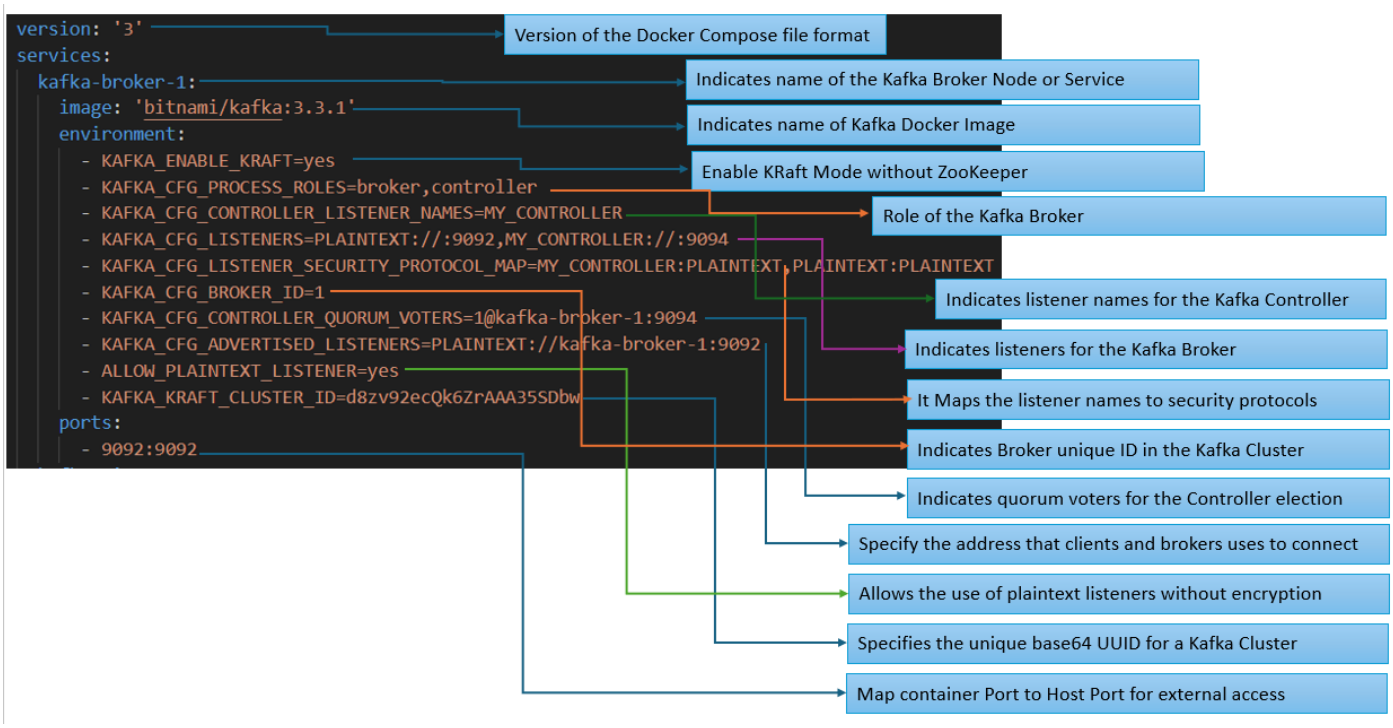
Heartbeat Messages:

- The leader sends regular heartbeat messages to followers to maintain authority.
- If a follower doesn't receive a heartbeat within a set time, it assumes the leader has failed and starts a new election.

10. Setting Up Kafka Locally with Docker and KRaft Mode

To set up Apache Kafka for development in KRaft mode using Docker

- Install Docker Desktop using https://docs.docker.com/desktop/
- Create a docker-compose.yml file with the following content



This configuration sets up a single Kafka broker running in KRaft mode

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAI
NTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-1:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
```

```
    ports:
      - 9092:9092
```

As per above docker-compose.yml file

```
version: '3'
```
Specifies the version of the Docker Compose file format

services:

Defines the services that will be part of the Docker Compose app. Each service represents a container.

```
kafka-broker-1:
```
Indicates name of service and also name of the Kafka broker container name

```
image: 'bitnami/kafka:3.3.1'
```
Specifies the name of the Kafka docker image used for this service

environment:
Defines environment variables that will be passed to the Kafka broker node or container

```
- KAFKA_ENABLE_KRAFT=yes
```
It Enables KRaft mode and allows Kafka to run without Zookeeper.

```
- KAFKA_CFG_PROCESS_ROLES=broker,controller
```
It specifies the roles that a Kafka node can perform. Here, it is configured to act as both a broker and a controller

Broker Role:
A broker handles client requests like producing and consuming messages, stores data on disk, manages topics, partitions, and logs, handles client connections, and replicates data across other brokers.

Controller Role:
A controller manages the metadata for the Kafka cluster, including brokers, topics, and partitions. It elects partition leaders, manages topic creation and deletion, and ensures metadata consistency across the cluster.

When a Kafka process is configured to act as both a broker and a controller, it handles both client requests and metadata management. This setup can simplify the deployment by reducing the number of separate processes needed

```
- KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
```

It specifies the names of the listeners that the Kafka controller will use to communicate with each other and with brokers.

The property is a comma-separated list of listener names(you can use any names). These names must match the listener names defined in the KAFKA_CFG_LISTENERS property

```
- KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
```

It defines the listeners for the Kafka brokers and specifies how it will accept connections from clients and other brokers.

It supports various protocols like PLAINTEXT, SSL, SASL and etc.

It supports various protocols like PLAINTEXT,SSL, SASL and etc

Format :
{Protocol}:{Host}://{Port}

Example:
PLAINTEXT:10.0.0.1//:9092 ---> listen for 10.0.0.1
PLAINTEXT://:9092 ---> listen for all available network interfaces

PLAINTEXT Listener:

This listener will accept plaintext (unencrypted) connections on port 9092. It is typically used for client connections (producers and consumers).

MY_CONTROLLER Listener:

This listener will accept connections on port 9094 specifically for controller communication. It is used for internal communication between Kafka controllers and brokers

```
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
```

It specifies which security protocol each listener will use. Here, both MY_CONTROLLER and PLAINTEXT listeners will use the plaintext protocol.

This setting ensures that each listener uses the appropriate security protocol.

MY_CONTROLLER:PLAINTEXT
This listener will use the plaintext protocol, meaning the communication is unencrypted. This listener is typically used for internal communication between Kafka controllers and brokers.

PLAINTEXT:PLAINTEXT

This listener will also use the plaintext protocol. This listener is typically used for client connections (producers and consumers) and broker-to-broker communication.

```
- KAFKA_CFG_BROKER_ID=1
```
Sets the broker ID to 1. Each broker in a Kafka cluster must have a unique ID

```
- KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
```
This is used in Apache Kafka's KRaft mode to define the set of controller nodes that participate in the quorum for voting to elect the Controller or Leader. Here, it specifies that the controller quorum consists of the broker itself, listening on port 9094.

Format: node_id@host:port

Example:
if you have three Kafka nodes acting as controllers, the configuration might look like this

KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094,2@kafka-broker-2:9094,3@kafka-broker-3:9094

The above specified nodes form the quorum. A majority of these nodes must agree on any changes to the cluster metadata

```
- KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-1:9092
```
It specifies the addresses that clients and other brokers used to connect to the given node.

This listener will advertise the address kafka-broker-1:9092 to clients and other brokers. Clients and brokers will use this address to connect to the Kafka broker

```
- ALLOW_PLAINTEXT_LISTENER=yes
```

This configuration allows Kafka to start with plaintext listeners defined in the KAFKA_CFG_LISTENERS property, which means that the communication between clients and the broker, as well as between brokers, is not encrypted.

This setting is typically used for development and testing environments

```
- KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
```

It is typically a base64 format of UUID used to identify the Kafka cluster.This ID is crucial for the KRaft mode, where Kafka operates without Zookeeper and uses a quorum-based consensus algorithm for metadata management.

This is important for ensuring that all nodes in the cluster recognize each other and can participate in the quorum.

It can be generated using **kafka-storage.sh** tool provided with Kafka
> bin/kafka-storage.sh random-uuid

You can also use below Online tool for generating random uuid
https://www.fileformat.info/tool/guid-base64.htm

```
    ports:
      - 9092:9092
```

Maps port 9092 on the host to port 9092 on the container, allowing external access to the Kafka broker

Format: host_port:container_port

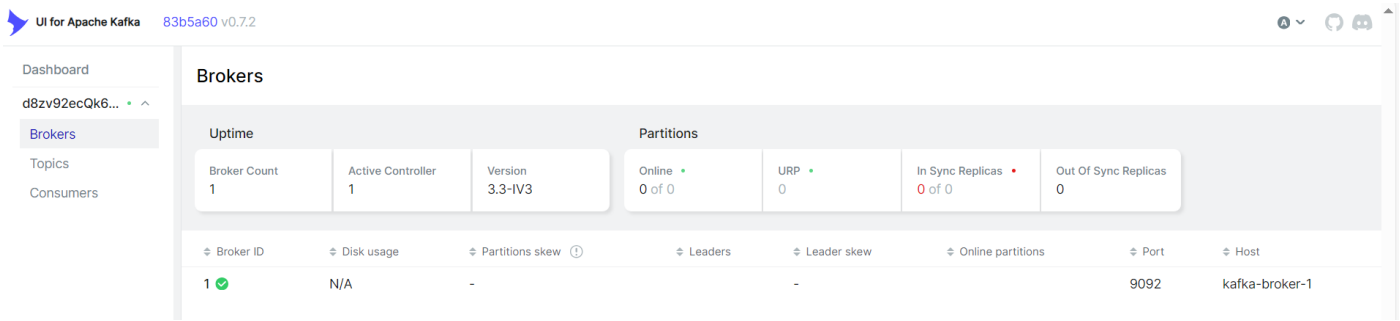
```
docker compose -f "docker-compose.yml" up -d --build
// run to build and start the container
```

Kafka UI Docker image

The Kafka UI Docker image from Provectus Labs makes it easy to set up a web-based interface for managing and monitoring your Kafka clusters.

Use the below configurations to start the Kafka UI docker container

```
  kafka-ui:
    container_name: kafka-ui
    image: 'provectuslabs/kafka-ui:latest'
    ports:
      - "8080:8080"
    environment:
      - KAFKA_CLUSTERS_0_BOOTSTRAP_SERVERS=kafka-broker-1:9092
      - KAFKA_CLUSTERS_0_NAME=d8zv92ecQk6ZrAAA35SDbw
```



Here's the complete Docker Compose file for a single Kafka broker running in KRaft mode

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAI
NTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-1:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
```

```
      - 9092:9092
  kafka-ui:
    container_name: kafka-ui
    image: 'provectuslabs/kafka-ui:latest'
    ports:
      - "8080:8080"
    environment:
      - KAFKA_CLUSTERS_0_BOOTSTRAP_SERVERS=kafka-broker-1:9092
      - KAFKA_CLUSTERS_0_NAME=d8zv92ecQk6ZrAAA35SDbw
```

Here's the complete Docker Compose file for multiple Kafka brokers running in KRaft mode

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094,2@kafka-broker-2:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-1:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
      - 9092:9092
  kafka-broker-2:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      - KAFKA_CFG_BROKER_ID=2
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094,2@kafka-broker-2:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-2:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
      - 9093:9092
```

```
  kafka-ui:
    container_name: kafka-ui
    image: 'provectuslabs/kafka-ui:latest'
    ports:
      - "8080:8080"
    environment:
      - KAFKA_CLUSTERS_0_BOOTSTRAP_SERVERS=kafka-broker-1:9092
      - KAFKA_CLUSTERS_0_NAME=d8zv92ecQk6ZrAAA35SDbw
```

```
docker compose -f "docker-compose.yml" up -d --build
// run to build and start the container
```

11. Creating and Managing Kafka Topics

Topic

A topic is a category where records are stored in the Kafka cluster. Topics are divided into multiple partitions, enabling parallel data processing.

Kafka topic names have a maximum length of **249** characters includes alphanumeric characters, periods (.), underscores (_), and hyphens (-). Therefore, it's important to keep them concise and meaningful.

An effective Kafka topic name can be constructed using the following components:

- **Domain or Entity** : Identifies the origin of the data, such as logs, product, user,hr, sales
- **Data Type or Action**: Specifies the nature of the data or event, such as update, click
- **Environment or Region**: Indicates the environment or geographic location, for example, "prod" or "dev" for environments, or "us-east" and "eu-west" for regions
- **Version**: Indicates version of the topic, such as v1, v2

Examples:

sales.orders.us-east.v1, sales.refunds.dev.v1, user.profile.update.eu-west.v1, user.login.failed, hr.leave.requests, logs.application.error

Partition

A topic divided into multiple partitions for scalability and fault tolerance. Each partition is an ordered, immutable sequence of records that is continuously appended to a commit log. Partitions allow Kafka to scale horizontally by distributing data across multiple servers.

Example

- If a topic has 3 partitions and there are 3 brokers, each broker will have one partition.
- If a topic has 3 partitions and there are 5 brokers, the first 3 brokers will each have one partition, while the remaining 2 brokers will not have any partitions for that specific topic.
- If a topic has 3 partitions and there are 2 brokers, each broker will share more than partition

Determining the right number of partitions for a Kafka topic generally depends on the total throughput for the topic, as well as the throughput per partition for both production and consumption.

Formula to calculate the minimum number of partitions

No.of Partitions = max(t/p, t/c)

t = target throughput for a topic(The total amount of data you want to handle per second)
p = measured throughput on a single production partition(i.e. how much data can be written to a partition per second)

c = measured throughput on a single consumption partition (i.e. how much data can be read from a partition per second)

Example 1:

Let's say, t=1000 messages/second, p=100 messages/second, c=200 messages/second

Number of Partitions=max(1000/100,1000/200) = 10

Example 2:

Let's say, t= 1000 MB/sec, p= 200 MB/sec, c= 150 MB/sec

Number of Partitions=max(1000/200,1000/150) = ~7

Replication Factor

The replication factor is set at the topic level when the topic is created. It specifies how many copies of each partition will be stored across different brokers in a Kafka cluster for fault tolerance and high availability.

Make sure the replication factor is less than or equal to the number of available brokers. otherwise, you'll encounter an "Invalid Replication Factor" error.

The ideal replication factor for Kafka topics is typically 3. This provides a good balance between redundancy and resource utilization. It can survive the failure of up to two brokers without any data loss.

```
/*  Create kafka topic
Syntax: kafka-topics.sh --create --topic <topic_name> --partitions <no_of_partitions>
--replication-factor <replication_factor> --bootstrap-server <kafka_broker>

--create: flag to create a topic
--topic: name of the topic
--partitions: number of partitions for the topic
--replication-factor: number of replicas for each partition
--bootstrap-server: kafka broker address to connect
*/
# Create kafka topic with 3 partitions and 1 replication factor
$ kafka-topics.sh --create --topic test-topic1 --partitions 3 --replication-factor 1 --bootstrap-server localhost:9092

/* List all topics
Syntax: kafka-topics.sh --list --bootstrap-server <kafka_broker>
--list: flag to list all topics
--bootstrap-server: kafka broker address to connect
*/
# List all topics
$ kafka-topics.sh --list --bootstrap-server localhost:9092

/*  Describe a topic
Syntax: kafka-topics.sh --describe --topic <topic_name> --bootstrap-server <kafka_broker>
--describe: flag to describe a topic
--topic: name of the topic
--bootstrap-server: kafka broker address to connect
*/
# Describe a topic
$ kafka-topics.sh --describe --topic test-topic1 --bootstrap-server localhost:9092

/*  Delete a topic
Syntax: kafka-topics.sh --delete --topic <topic_name> --bootstrap-server <kafka_broker>
--delete: flag to delete a topic
--topic: name of the topic
--bootstrap-server: kafka broker address to connect
*/
# Delete a topic
$ kafka-topics.sh --delete --topic test-topic1 --bootstrap-server localhost:9092
```

Create the following docker-compose.yml to sets up a single Kafka broker in the KRaft mode

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAI
NTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-1:9092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
      - 9092:9092
```

Run the below command to build and start the docker container

```
docker compose -f "docker-compose.yml" up -d --build
// run to build and start the container
```

```
// View list of docker processes
$ docker ps
CONTAINER
ID   IMAGE                          COMMAND                  CREATED       STA
TUS       PORTS
bf76f525a3b8   bitnami/kafka:3.3.1            "/opt/bitnami/script…"   4 hours
ago   Up 4 hours   0.0.0.0:9092->9092/tcp
```

enter docker container to create Kafka topic using CLI

```
// enter the container
// Syntax: docker exec -it <container_id> <command>
$ docker exec -it bf76f525a3b8 bin/sh
```

Create Kafka Topic

```
// create kafka topic

/*
Syntax: kafka-topics.sh --create --topic <topic_name> --partitions
<no_of_partitions> --replication-factor <replication_factor> --bootstrap-server
<kafka_broker>

--create: flag to create a topic
--topic: name of the topic
--partitions: number of partitions for the topic
--replication-factor: number of replicas for each partition
--bootstrap-server: kafka broker address to connect
*/
$ kafka-topics.sh --create --topic test-topic1 --partitions 3 --replication-
factor 1 --bootstrap-server localhost:9092

Created topic test-topic1.
```

```
// as there is only one broker running, replication factor should be 1. It will
throw an error if replication factor is more than the number of brokers
$ kafka-topics.sh --create --topic test-topic2 --partitions 3 --replication-
factor 2 --bootstrap-server localhost:9092

Error while executing topic command : Unable to replicate the partition 2
time(s): The target replication factor of 2 cannot be reached
because only 1 broker(s) are registered.
```

Create Kafka Topic With Custom Configuration

```
$ kafka-topics.sh --create --topic test-topic3 --partitions 3 --replication-
factor 1 --bootstrap-server localhost:9092 \
  --config retention.ms=604800000 \
  --config cleanup.policy=compact \
  --config min.insync.replicas=2 \
  --config segment.bytes=1073741824 \
  --config compression.type=gzip \
  --config max.message.bytes=1048576 \
  --config message.timestamp.type=CreateTime

  Explaination of Settings:

    topic: The name of the topic
    partitions: The number of partitions for the topic
```

```
    replication-factor: The number of replicas for each partition across
different brokers
    bootstrap-server: The Kafka broker address to connect
    retention.ms: Defines how long Kafka retains messages in the topic. This
can be set by time or size(default is 7 days)
    cleanup.policy: Determines how old data is cleaned up. Options include
delete/compact(default is delete)
    min.insync.replicas: Minimum number of replicas that must acknowledge a
write to be considered successful(default is 1)
    segment.bytes: Size of the log segment files Ex:1073741824 (default is 1
GB).
    compression.type: Type of compression for the messages like gzip, snappy,
lz4, etc (default is none)
    max.message.bytes: The maximum size of a message that can be sent to the
topic(default is 1 MB)
    message.timestamp.type: Specifies whether the timestamp in the message is
set by the producer(CreateTime)
                            or the broker(LogAppendTime) (default is
CreateTime)
```

List all Topics

```
// list all topics
/*
Syntax: kafka-topics.sh --list --bootstrap-server <kafka_broker>

--list: flag to list all topics
--bootstrap-server: kafka broker address to connect
*/
$ kafka-topics.sh --list --bootstrap-server localhost:9092

test-topic1
```

Describe a Topic

```
// describe a topic to get details about the a specific topic
/*
Syntax: kafka-topics.sh --describe --topic <topic_name> --bootstrap-server
<kafka_broker>

--describe: flag to describe a topic
--topic: name of the topic
--bootstrap-server: kafka broker address to connect
*/
$ kafka-topics.sh --describe --topic test-topic1 --bootstrap-server localhost:9092
```

```
Topic: test-topic1      TopicId: K-1z_KKtQfqb5TRSSHIHyA PartitionCount:
3       ReplicationFactor: 1    Configs:
        Topic: test-topic1      Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: test-topic1      Partition: 1    Leader: 1       Replicas: 1     Isr: 1
        Topic: test-topic1      Partition: 2    Leader: 1       Replicas: 1     Isr: 1
```

Alter a Topic

```
// alter a topic to increase partitions
/*
Syntax: kafka-topics.sh --alter --topic <topic_name> --partitions
<no_of_partitions> --bootstrap-server <kafka_broker>

--alter: flag to alter a topic
--topic: name of the topic
--partitions: number of partitions for the topic
--bootstrap-server: kafka broker address to connect
*/
$ kafka-topics.sh --alter --topic test-topic1 --partitions 5 --bootstrap-server
localhost:9092

WARNING: If partitions are increased for a topic that has a key, the partition
logic or ordering of the messages will be affected
partitions for topic test-topic1 increased to 5.
```

Delete a Topic

```
// delete a topic
/*
Syntax: kafka-topics.sh --delete --topic <topic_name> --bootstrap-server
<kafka_broker>

--delete: flag to delete a topic
--topic: name of the topic
--bootstrap-server: kafka broker address to connect
*/
$ kafka-topics.sh --delete --topic test-topic1 --bootstrap-server
localhost:9092
Topic test-topic1 is marked for deletion.
```

12. Setting Up a Kafka Producer in Node.js using KafkaJS

Producers are clients that publish messages to Kafka topics, distributing them across various partitions. They send data to the broker, which then stores it in the corresponding partition of the topic.

Each message or record that a producer sends includes a Key (optional), Value, Header (optional), and Timestamp.

For additional details about Kafka Producer, please refer to the following link
Apache Kafka for Developers #4: Kafka Producer and Acknowledgements

Prerequisites:

- Set up a Kafka Cluster by following the link Apache Kafka for Developers #10:Setting Up Kafka Locally with Docker
- Set up a Kafka Topic by following the link Apache Kafka for Developers #11: Creating and Managing Kafka Topics

create a separate listener called EXTERNAL_HOST for the producer app running outside the docker environment

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      -
KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094,EXTERNAL_HOST://:29092
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,E
XTERNAL_HOST:PLAINTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-
1:9092,EXTERNAL_HOST://localhost:29092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
      - 29092:29092
```

install KafkaJs library using

```
npm install kafkajs
```

Publish messages to Kafka using a Message Key

Message key determines the partition to which a message is sent, ensuring that all messages with the same key are routed to the same partition. This helps maintain the order of related messages and allows for efficient processing.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerWithKey = async () => {
    try {
        // Connect to the Kafka broker
        await producer.connect();

        // Send a message to the topic 'test-topic1'
        await producer.send({
            topic: 'test-topic1',
            messages: [{ key: 'key1', value: 'first message' }],
        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
        await producer.disconnect();
    }
};

// Run the producer
ProducerWithKey().catch(console.error);
```

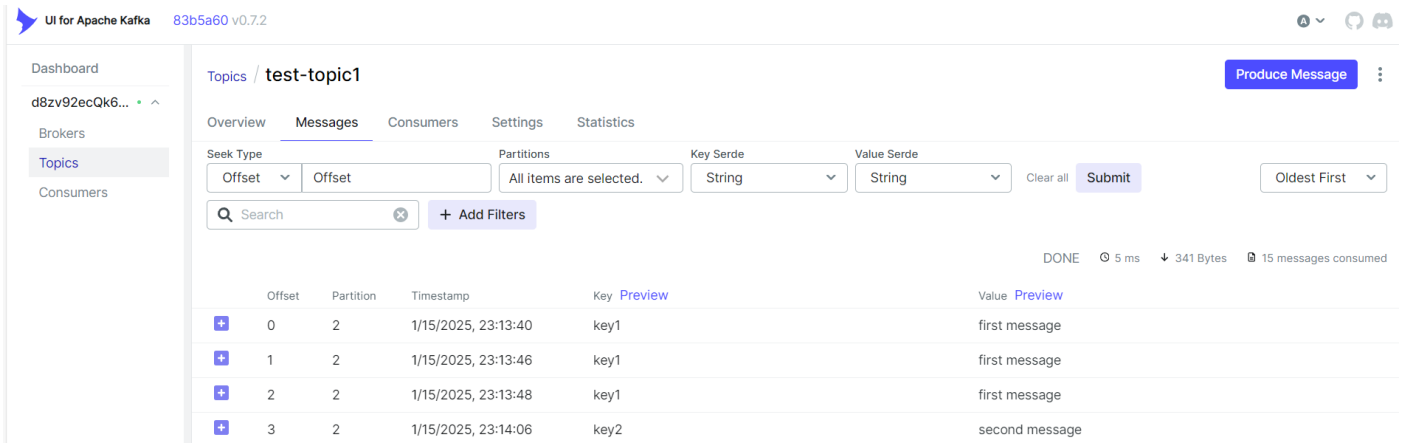
modify "scripts" as below in the package.json

```
 "scripts": {
    "producer-start": "node producer.js"
  },
```

run the producer using

```
$ npm run producer-start
```

View messages using kafka-ui docker container



Publish messages to Kafka without Message Key

When publishing messages to Kafka without a key, they are distributed across partitions in a round-robin manner. This ensures an even load distribution but does not maintain the order of messages across partitions.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerWithoutKey = async () => {
    try {
        // Connect to the Kafka broker
        await producer.connect();

        // Send a message to the topic 'test-topic1'
```

```
        await producer.send({
            topic: 'test-topic1',
            messages: [{ value: 'first message without key' }],
        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
        await producer.disconnect();
    }
};

// Run the producer
ProducerWithoutKey().catch(console.error);
```

Publish messages to Kafka with Specific Partition

When publishing messages to a specific partition in Kafka, you can specify the partition number directly. This ensures that all messages are sent to the chosen partition, maintaining their order and enabling focused processing.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerToSpecificPartition = async () => {
    try {
        // Connect to the Kafka broker
        await producer.connect();

        // Send a message to the topic 'test-topic1' with partition 0
        await producer.send({
            topic: 'test-topic1',
            messages: [{ value: 'first message partition 0', partition: 0 }],
        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
```

```
        await producer.disconnect();
    }
};

// Run the producer
ProducerToSpecificPartition().catch(console.error);
```

Publish messages to Multiple Topics

When publishing messages to multiple Kafka topics, you can either use a single producer to send messages to different topics or create separate producers for each topic.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerToMultipleTopics = async () => {
    try {
        // Create an array of messages to send to multiple topics
        const messages = [
            { topic: 'test-topic1', messages: [{ value: 'message1' }, { value:
'message2' }] },
            { topic: 'test-topic2', messages: [{ value: 'message3' }, { value:
'message4' }] }
        ];

        // Connect to the Kafka broker
        await producer.connect();

        // Send the messages to the topics
        await producer.sendBatch({
            topicMessages: messages
        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
        await producer.disconnect();
    }
};
```

```
// Run the producer
ProducerToMultipleTopics().catch(console.error);
```

Publish messages with Headers

When publishing messages to Kafka with headers, you can include additional metadata as key-value pairs.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerWithHeader = async () => {
    try {
        // Connect to the Kafka broker
        await producer.connect();

        // Send a message to the topic 'test-topic1' with headers
        await producer.send({
            topic: 'test-topic1',
            messages: [{
                value: 'first message partition 0',
                partition: 0,
                headers: {
                    'correlation-id': '1234',
                    'client-id': 'my-test-app'
                }
            }]

        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
        await producer.disconnect();
    }
};

// Run the producer
ProducerWithHeader().catch(console.error);
```

Publish messages with Acknowledgements

Acknowledgements (acks) in Kafka are a mechanism to ensure that messages are reliably stored in the Kafka topic partition.

Kafka producers only write data to the leader partition in the broker.

Kafka producers must also set the acknowledgment level (acks) to indicate whether a message needs to be written to a minimum number of replicas before it is considered successfully written.

Control the number of required acks.

-1 = all insync replicas(min.insync.replicas property) must acknowledge (default)
0 = no acknowledgments
1 = only waits for the leader to acknowledge

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-test-app',
    brokers: ['localhost:29092']
});

// Create Kafka producer instance
const producer = kafka.producer();

const ProducerWithAcks = async () => {
    try {
        // Connect to the Kafka broker
        await producer.connect();

        // Send a message to the topic 'test-topic1' with acks
        await producer.send({
            topic: 'test-topic1',
            messages: [{ key: 'key1', value: 'first message' }],
            acks: 1
        });
    } catch (error) {
        console.error('Error occurred while sending the message', error);
    } finally {
        // Disconnect the producer
        await producer.disconnect();
    }
};
```

```
// Run the producer
ProducerWithAcks().catch(console.error);
```

13. Setting Up a Kafka Consumer in Node.js using KafkaJS

Consumers are clients that read data from Kafka topics. They subscribe to one or more topics and process the data. Each consumer keeps track of its position in each partition using partition offset.

Kafka consumers follow a pull model. This means that consumers actively request data from Kafka brokers rather than having the brokers push data to them

Consumer Group

A group of consumers work together to consume data from a topic. Each consumer in the group processes data from different partitions, allowing for parallel processing and load balancing.

Kafka consumers are typically part of a consumer group. When multiple consumers are subscribed to a topic and are part of the same group, each consumer will receive messages from a different subset of the partitions in the topic. Consumer groups must have unique group ids within the cluster

For additional details about Kafka Consumer, please refer to the following link
Apache Kafka for Developers #5: Kafka Consumer and Consumer Group

Prerequisites:

- Set up a Kafka Cluster by following the link Apache Kafka for Developers #10:Setting Up Kafka Locally with Docker
- Set up a Kafka Topic by following the link Apache Kafka for Developers #11: Creating and Managing Kafka Topics
- Produce messages into Kafka Topic by following the link Apache Kafka for Developers #12: Setting Up a Kafka Producer in Node.js using KafkaJS

create a separate listener called EXTERNAL_HOST for the producer app running outside the docker environment

```
version: '3'
services:
  kafka-broker-1:
    image: 'bitnami/kafka:3.3.1'
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=MY_CONTROLLER
      -
KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,MY_CONTROLLER://:9094,EXTERNAL_HOST://:29092
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=MY_CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,E
XTERNAL_HOST:PLAINTEXT
      - KAFKA_CFG_BROKER_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka-broker-1:9094
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka-broker-
1:9092,EXTERNAL_HOST://localhost:29092
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_KRAFT_CLUSTER_ID=d8zv92ecQk6ZrAAA35SDbw
    ports:
      - 29092:29092
```

install KafkaJs library using

```
npm install kafkajs
```

Set up a Kafka consumer with auto commit enabled

```
// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-consumer-app',
    brokers: ['localhost:29092']
});

// Create a new consumer instance
const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const ConsumerAutoCommitSingleTopic = async () => {
    // Connecting the consumer
    await consumer.connect();
```

```
    // Subscribing to the topic and set fromBeginning to true
    await consumer.subscribe({ topic: 'test-topic1', fromBeginning: true });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                topic, partition, offset: message.offset, key: (message.key ||
''').toString(), value: message.value.toString()
            });
        },
    });
};

// Run the consumer
ConsumerAutoCommitSingleTopic().catch(console.error);

process.on('unhandledRejection', async e => {
    try {
        console.error(e);
        await consumer.disconnect()
        process.exit(0)
    } catch (_) {
        process.exit(1)
    }
})
process.on('uncaughtException', async e => {
    try {
        console.error(e);
        await consumer.disconnect()
        process.exit(0)
    } catch (_) {
        process.exit(1)
    }
})

const signals =[ 'SIGTERM', 'SIGINT', 'SIGUSR2']

signals.forEach(type => {
    process.once(type, async () => {
        try {
            console.log(`Received signal: ${type}`)
            await consumer.disconnect()
        } finally {
            process.kill(process.pid, type)
        }
    })
})
```

modify "scripts" as below in the package.json

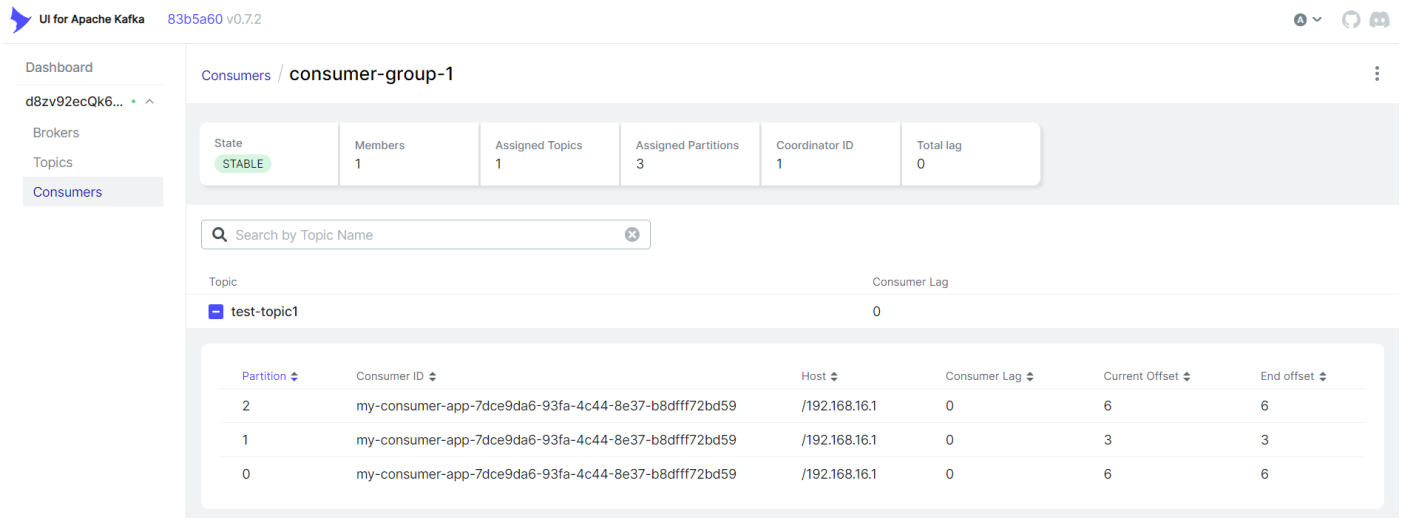
```
  "scripts": {
    "consumer-start": "node consumer.js"
  },
```

run the consumer using

```
$ npm run consumer-start
```

Since there is only one consumer in the group, Consumer 1 will be assigned all three partitions of Topic(test-topic1) . This means Consumer 1 will read messages from Partition 0, Partition 1, and Partition 2.

View consumer details using kafka-ui docker container



Set up a Kafka consumer with Multiple Topics

```
// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-consumer-app',
    brokers: ['localhost:29092']
});

// Create a new consumer instance
const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
```

```
});

const ConsumerAutoCommitMultiTopic = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic and set fromBeginning to true
    await consumer.subscribe({ topic: ['test-topic1', 'test-topic2'],
fromBeginning: true });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                topic, partition, offset: message.offset, key: (message.key ||
'').toString(), value: message.value.toString()
            });
        },
    });
};

// Run the consumer
ConsumerAutoCommitMultiTopic().catch(console.error);
```

Set up a Kafka consumer with Manual Commit

```
// Create a consumer that reads from the beginning of the topic
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-consumer-app',
    brokers: ['localhost:29092']
});

// Create a new consumer instance
const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: false
});

const ConsumerManualCommit = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic and set fromBeginning to true
    await consumer.subscribe({ topic: 'test-topic1', fromBeginning: true });

    // Running the consumer
    await consumer.run({
```

```
        eachMessage: async ({ topic, partition, message }) => {
            console.log({
                topic, partition, offset: message.offset, key: (message.key ||
'').toString(), value: message.value.toString()
            });

            // Synchronous commit: Waits for the commit to complete
            await consumer.commitOffsets([
                { topic, partition, offset: (parseInt(message.offset, 10) +
1).toString() }
            ]);
        },
    });
};

// Run the consumer
ConsumerManualCommit().catch(console.error);
```

Set up a Kafka consumer to consume Latest messages

In this setting(fromBeginning: false), the consumer begins reading from the latest message upon subscribing to the topic partition. This setting is ideal when you only need to process new messages.

```
const { Kafka } = require('kafkajs');

// Initialize a new Kafka instance with the client ID and broker address
const kafka = new Kafka({
    clientId: 'my-consumer-app',
    brokers: ['localhost:29092']
});

// Create a new consumer instance
const consumer = kafka.consumer({
    groupId: 'consumer-group-1',
    autoCommit: true,
    autoCommitInterval: 5000
});

const ConsumerReadMessageFromLatest = async () => {
    // Connecting the consumer
    await consumer.connect();

    // Subscribing to the topic and set fromBeginning to false
    await consumer.subscribe({ topic: 'test-topic1', fromBeginning: false });

    // Running the consumer
    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
```

```
            console.log({
                topic, partition, offset: message.offset, key: (message.key ||
'').toString(), value: message.value.toString()
            });
        },
    });
};

// Run the consumer
ConsumerReadMessageFromLatest().catch(console.error);
```

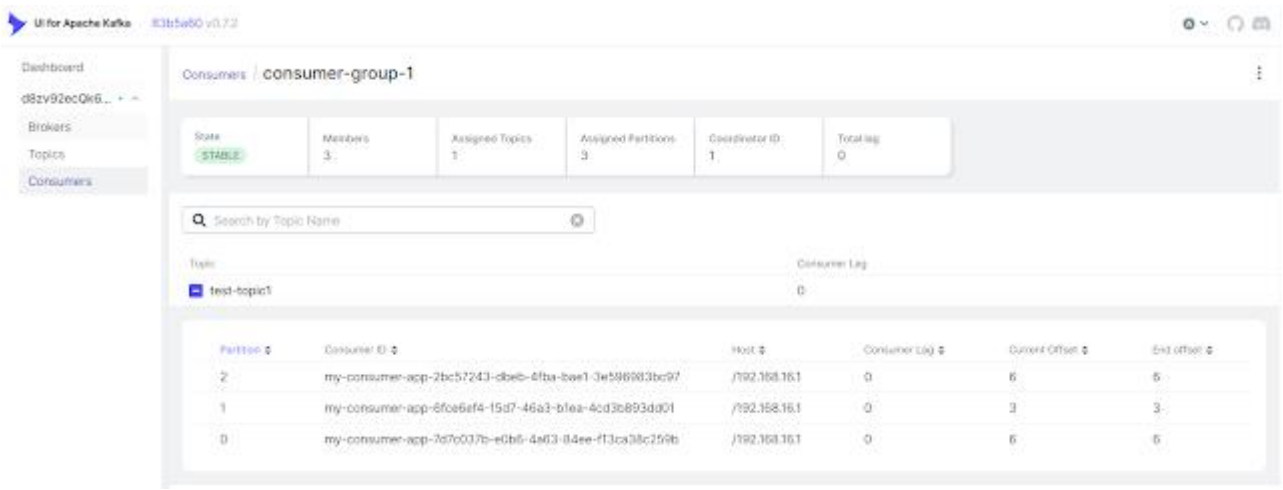
Set up a Consumer group with three consumers

With this setup, we can create multiple instances of consumers with the same group ID. Each consumer will be part of the same group and will share the load of consuming messages from the topic.

```
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
    clientId: 'my-consumer-app',
    brokers: ['localhost:29092']
});

const createConsumer = async (groupId, instanceId) => {
    const consumer = kafka.consumer({ groupId, autoCommit: true,
autoCommitInterval: 5000 });

    await consumer.connect();
    await consumer.subscribe({ topic: 'test-topic1', fromBeginning: true });

    await consumer.run({
        eachMessage: async ({ topic, partition, message }) => {
            console.log(`Consumer ${instanceId} -
${message.value.toString()}`);
        },
    });

    return consumer;
};

const multiConsumer = async () => {
    const groupId = 'consumer-group-1';
    const consumers = [];

    // Create 3 consumers
    for (let i = 0; i < 3; i++) {
        consumers.push(createConsumer(groupId, i));
    }
    await Promise.all(consumers);
```

```
};

// Graceful shutdown
process.on('SIGINT', async () => {
    console.log('SIGINT received, shutting down...');
    await Promise.all(consumers.map(consumer => consumer.disconnect()));
    process.exit(0);
});

process.on('SIGTERM', async () => {
    console.log('SIGTERM received, shutting down...');
    await Promise.all(consumers.map(consumer => consumer.disconnect()));
    process.exit(0);
});

multiConsumer().catch(console.error);
```

Since there are three partitions and three consumers, each consumer will be assigned exactly one partition. It means Consumer 1 might be assigned Partition 0, Consumer 2 might be assigned Partition 1 and Consumer 3 might be assigned Partition 2



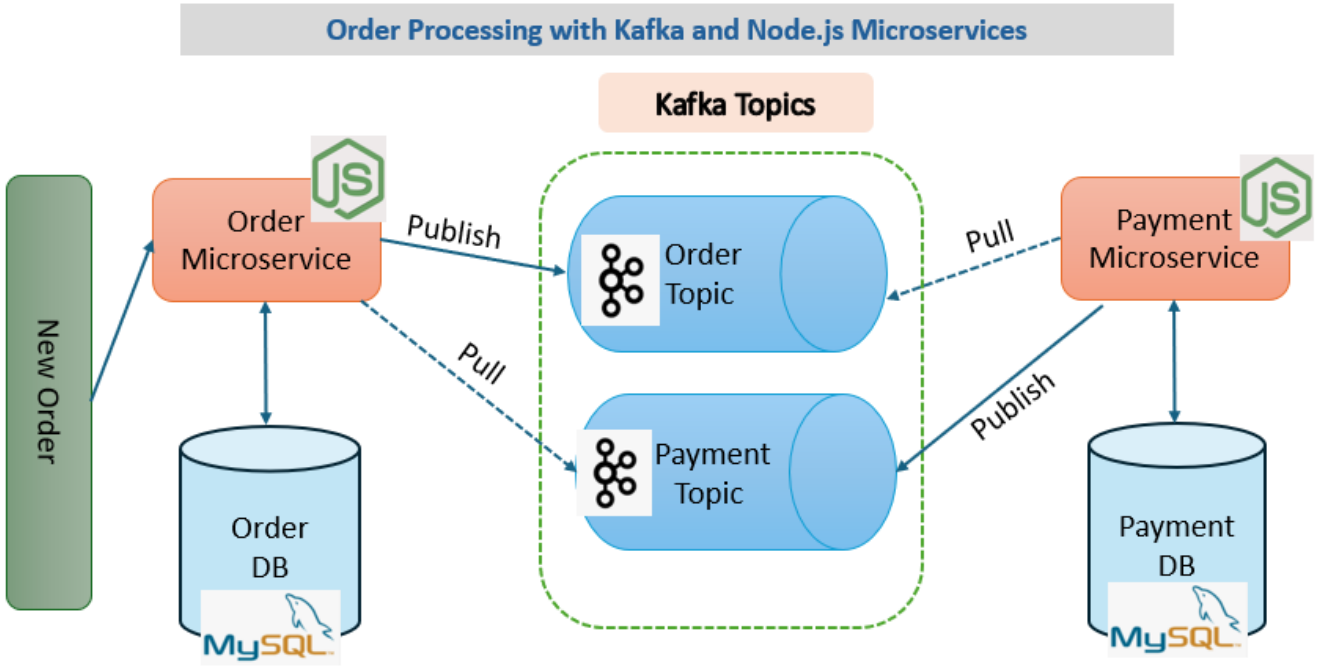
In a production environment, it's recommended to use a Kubernetes deployment with multiple replicas to run each Kafka consumer in separate containers. This approach provides scalability and fault tolerance.

You can explore everything about Kubernetes in detail @https://millionvisit.blogspot.com/search/label/kubernetes

14. Order Processing with Kafka, Node.js Microservices, and MySQL

Building simple E-commerce Order Processing System with Kafka, Nodejs Microservices, and MySQL Databases.

Architecture Overview:



The above architecture consists of two main Node.js microservices

Order Microservice:

The Order Microservice is responsible for accepting new order requests, storing the details in the Order Database, and publishing the order information to the Kafka Order Topic.

Implementing Idempotency Checks

Idempotency checks are crucial to prevent the duplicate processing of the same request. The Order Microservice ensures that each request header contains a unique identifier called the **idempotency-key**. This key is stored in the Order Database along with the order details. Before processing a request, the service checks the database to determine if the request is a duplicate, ensuring that each order is processed only once.

Implementing Kafka Retry

The Order Microservice includes Kafka retry logic to handle transient errors. By using try-catch blocks to capture exceptions, the service can reprocess messages up to a maximum number of retries with a delay between each attempt. This ensures that temporary issues do not prevent order processing service.

Subscribing to Kafka Payment Topic

The Order Microservice subscribes to the Kafka Payment Topic to receive payment status updates. This allows the Order Microservice to update the order status in the Order Database based on the payment status received from the Payment Microservice.

Payment Microservice

The Payment Microservice listens to the Kafka Order Topic for new orders, processes payments, and stores payment statuses in a database, and publishing the payment information to the Kafka Payment Topic.

Implementing Idempotency Checks

To ensure idempotency, the Payment Microservice checks if a payment record with the same orderId already exists in the Payment Database before processing the payment. This prevents duplicate processing of the same order.

Publishing Payment Status to Kafka Payment Topic

After processing the payment, the Payment Microservice publishes the payment status to the Kafka Payment Topic. This allows the Order Microservice to receive updates on the payment status and update the order status accordingly.

Step-by-Step Implementation

Prerequisites

- Install Docker Desktop using https://docs.docker.com/desktop/

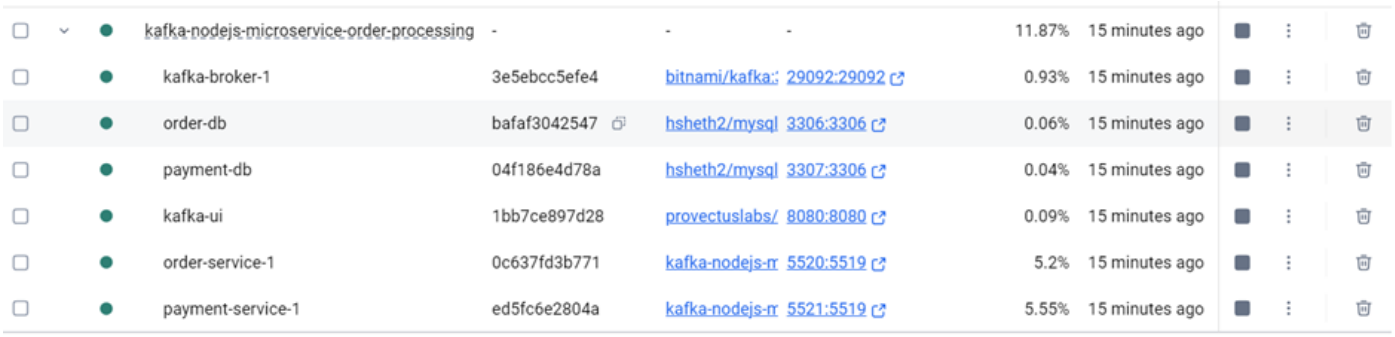
1. Clone the repository

https://github.com/ramasubbareddy1224/kafka-nodejs-microservice-order-processing

```
> git clone https://github.com/ramasubbareddy1224/kafka-nodejs-microservice-order-processing.git

// change directory
> cd kafka-nodejs-microservice-order-processing
```

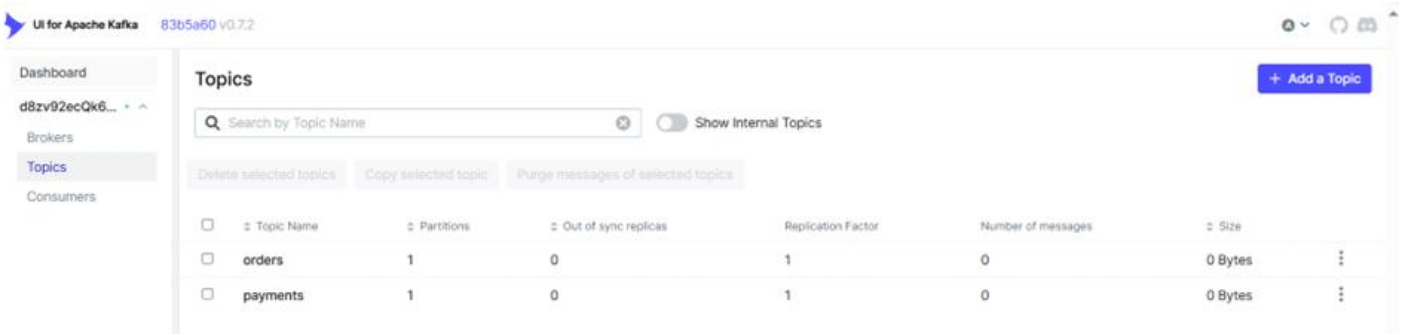
2. Start all the Docker containers by running the following command.

```
> docker-compose up -d --build
```



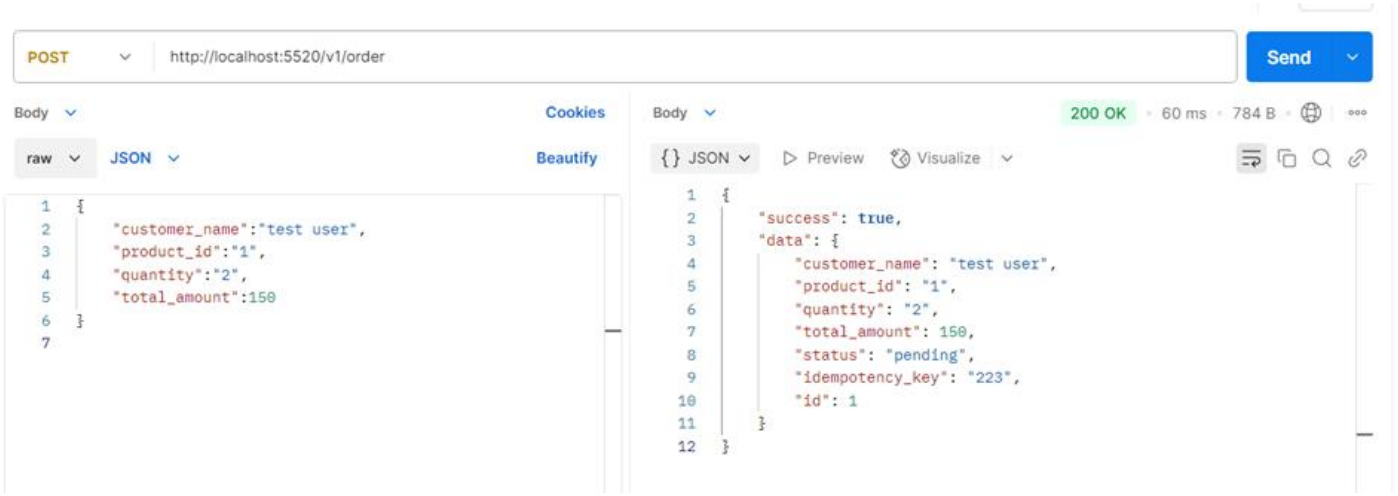
3. Use Kafka UI to view the list of Kafka topics.

http://localhost:8080/ui/clusters/d8zv92ecQk6ZrAAA35SDbw/all-topics

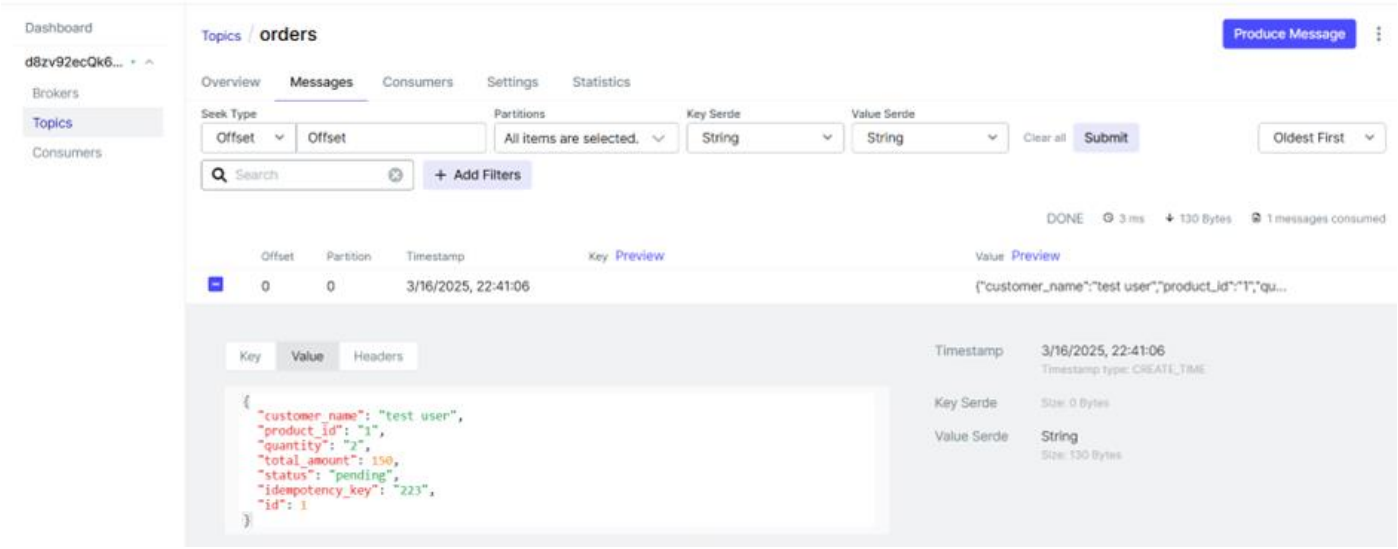


4. Submit a new order request using either curl or Postman

```
curl --location 'http://localhost:5520/v1/order' \
--header 'idempotency-key: 223' \
--header 'Content-Type: application/json' \
--data '{
    "customer_name":"test user",
    "product_id":"1",
    "quantity":"2",
    "total_amount":150
}'
```



5. Verify the message in the Kafka Order Topic



6. Review the details in the Order Database

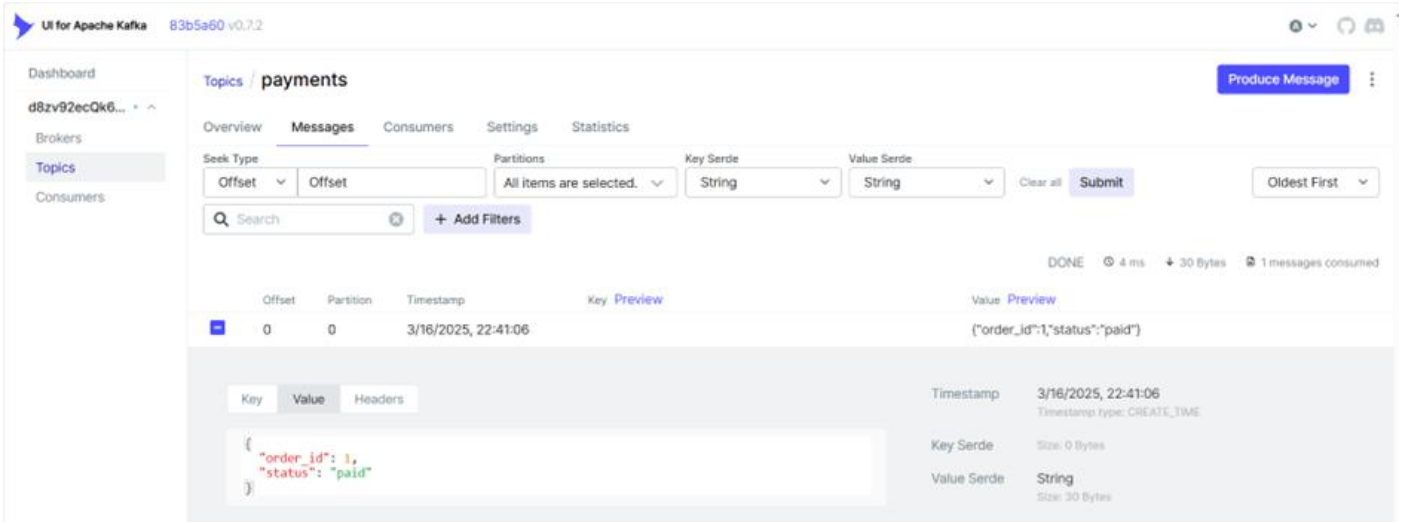
```
1 SELECT id, customer_name, product_id, quantity, total_amount, status, idempotency_key, created_at, updated_at
2 FROM order_db.orders;
```

orders 1

SELECT id, customer_name, product_id, quantity, total | Enter a SQL expression to filter results (use Ctrl+Space)

	id	customer_name	product_id	quantity	total_amount	status	idempotency_key	created_at	updated_at
1	1	test user	1	2	150	paid	223	2025-03-16 17:11:06	2025-03-16 17:11:06

7. Verify the message in the Kafka Payment Topic.



8. Review the details in the Payment Database.

```
1 SELECT id, order_id, status, created_at, updated_at
2 FROM payment_db.payments;
```

payments 1

SELECT id, order_id, status, created_at, updated_at FR | Enter a SQL expression to filter results (use Ctrl+Space)

	id	order_id	status	created_at	updated_at
1	1	1	paid	2025-03-16 17:11:06	2025-03-16 17:11:06