

# Kafka Internal Consumer ConsumerGroup



How does consumer pull messages from broker in Kafka. Is it on the basis of topic level or partition level



In Kafka, consumers pull messages from the broker at the **partition level**, not the topic level. Here's a breakdown of how this works:

## Key Concepts

### 1. Topic and Partitions:

- A topic in Kafka is divided into one or more partitions.
- Each partition is a log where messages are stored in an ordered sequence.

### 2. Consumer Groups:

- Consumers belong to a consumer group.
- Each partition is assigned to one consumer within a group, ensuring that no two consumers in the same group process the same partition.

### 3. Offsets:

- Each consumer tracks its progress in a partition using offsets.
- This offset determines which messages have been consumed.

## How Consumers Pull Messages

### • Partition Assignment:

- The consumer group coordinator assigns partitions to consumers within the group.
- Each consumer gets one or more partitions to read from.

### • Pull Mechanism:

- The consumer sends a **Fetch request** to the broker for the assigned partitions.
- The request includes details such as:

- The topic and partition.
- The offset from which to start reading.
- The maximum number of bytes to fetch.
- **Response:**
  - The broker responds with messages from the requested partitions, starting from the specified offset.

## Key Points:

- The pull mechanism operates at the **partition level**, meaning a consumer requests messages from specific partitions of a topic.
- Consumers do not pull messages for the entire topic unless they are assigned all its partitions.
- The Kafka client library abstracts much of this complexity, making it seem seamless to developers, but the underlying fetch operations are partition-specific.



can consumer pull messages from topic



No, **Kafka consumers cannot directly pull messages from a topic as a whole**. Consumers in Kafka operate at the **partition level**. However, since partitions are part of a topic, the consumer indirectly interacts with the topic by pulling messages from its partitions.

## Why Can't Consumers Directly Pull from a Topic?

### 1. Kafka's Design:

- A topic is a logical abstraction that consists of multiple partitions.
- Messages are stored and fetched at the partition level for scalability and parallelism.

### 2. Partition Assignment:

- In a Kafka cluster, partitions are distributed across brokers.
- To ensure efficient processing, each consumer pulls data from one or more specific partitions of the topic, not the entire topic.

## How Does a Consumer Work with a Topic?

When a consumer subscribes to a topic:

### 1. Partition Assignment:

- Kafka assigns partitions of the topic to the consumer (or to multiple consumers in a group).
- This assignment is managed by the Kafka consumer group coordinator.

### 2. Pull from Partitions:

- The consumer sends fetch requests to pull messages from the assigned partitions of the topic.

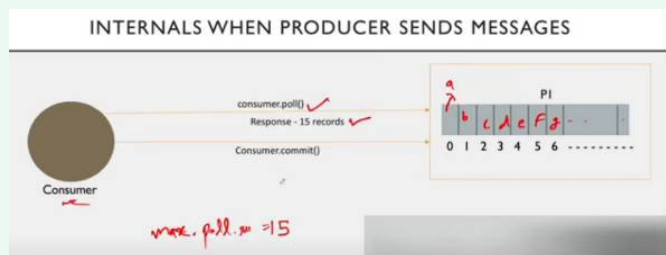
## Use Case: Simulating Topic-Level Consumption

If you want to simulate "pulling from a topic":

- You can subscribe to a topic using the Kafka consumer API.
- The Kafka client handles partition assignment for you, distributing partitions among consumers in a group.
- The consumer internally fetches messages from each partition of the topic it is assigned to.

## Summary

While you can't pull messages directly from a topic, subscribing to a topic allows you to interact with its partitions. This design ensures Kafka's scalability and supports distributed, parallel message processing.



Kindly explain the above image. What is the significance of current and committed offset set here



The diagram represents how a Kafka consumer interacts with a partition during the process of consuming messages. Here's a breakdown of the concepts and the significance of the **current offset** and **committed offset**:

---

## Key Elements in the Diagram:

### 1. Partition (PI):

- Represents a log of messages stored sequentially with offsets.
- Offsets (e.g., 0, 1, 2, ...) are unique identifiers for each message in the partition.

### 2. Consumer:

- Fetches messages from the partition by polling the broker.

### 3. Consumer Poll (`consumer.poll()`):

- The consumer requests a batch of messages from the assigned partition.
- In the diagram, the `poll` operation fetched 15 records (messages `a` to `o`).

### 4. Consumer Commit (`consumer.commit()`):

- After processing the fetched messages, the consumer commits the **offset** to Kafka.
  - Committing the offset indicates that all messages up to that offset have been processed successfully.
- 

## Significance of Current Offset and Committed Offset:

### 1. Current Offset:

- Refers to the **next offset** the consumer will read from the partition.
- After a `poll()`, the current offset moves forward to the next unread message in the partition.
- For example:
  - If the consumer just read messages `a` to `o` (offsets 0 to 14), the **current offset** will now be 15.
- This offset is managed in-memory by the consumer and is not saved until a commit is made.

### 2. Committed Offset:

- Refers to the last offset that the consumer has **successfully processed and committed** to Kafka.
- Kafka uses the committed offset to determine where to resume if the consumer restarts or fails.
- For example:
  - If the consumer commits offset 14, it indicates that messages up to offset 14 have been processed.
  - If the consumer crashes, it will resume from offset 15 (committed offset + 1).

## Workflow:

### 1. Fetching Messages:

- The consumer calls `poll()` and receives messages starting from the current offset.
- Example:
  - Current offset = 0.
  - After polling, the consumer receives messages `a` to `o`.

### 2. Processing Messages:

- The consumer processes the messages it has fetched.

### 3. Committing Offsets:

- After successfully processing messages, the consumer calls `commit()`.
- The committed offset is updated to reflect the last processed message (e.g., offset 14 in this case).

### 4. Next Fetch:

- The current offset moves forward (e.g., to 15) and the consumer is ready to poll more messages.

## Why are Current and Committed Offsets Important?

### 1. Fault Tolerance:

- The committed offset ensures that, in case of a consumer crash, the processing resumes from the last committed point, avoiding message loss or reprocessing.

### 2. Flow Control:

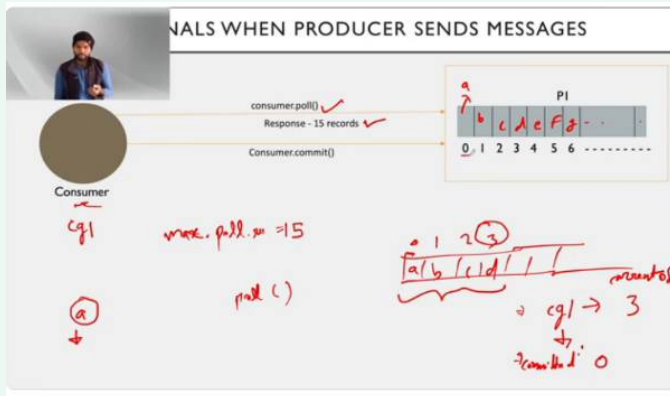
- The current offset allows the consumer to keep track of where it is in the partition.

### 3. Parallelism and Scalability:

- By managing offsets per partition, Kafka ensures that multiple consumers can process different partitions in parallel efficiently.

In summary, the **current offset** is the next message the consumer will read, while the **committed offset** represents the last successfully processed message. These offsets are critical for maintaining data integrity, fault tolerance, and efficient processing in Kafka.





Kindly explain the above image. Can the commit offset be greater than the current offset. What are the different types of acknowledgment while committing a message?



The diagram illustrates the interaction between a Kafka consumer and a partition, specifically focusing on polling messages, managing offsets, and committing progress. Here's a detailed explanation of the diagram, followed by answers to the questions:

## Explanation of the Diagram

### 1. Polling Messages (`consumer.poll()`):

- The consumer pulls messages from the Kafka broker for the assigned partition.
- For example, messages ``a``, ``b``, ``c``, and ``d`` are fetched sequentially, starting at offset ``0``.
- The `max.poll.records = 15`` indicates the maximum number of records the consumer can fetch in a single poll.

### 2. Current Offset:

- The current offset represents the **next message** the consumer will read.
- After consuming messages ``a``, ``b``, ``c``, and ``d`` (offsets ``0``, ``1``, ``2``, and ``3``), the current offset is ``4`` because the next unread message starts at offset ``4``.

### 3. Committed Offset:

- The committed offset is the last offset that the consumer has successfully processed and **committed**.
- In this diagram, the committed offset is ``0``, meaning only the message at offset ``0`` (``a``) has been processed and committed so far.

- The committed offset is maintained by Kafka to provide fault tolerance. If the consumer crashes, it will resume from the committed offset (`0` in this case).

---

## Can the Committed Offset Be Greater Than the Current Offset?

No, the **committed offset cannot be greater than the current offset** because:

1. The current offset is always the point from which the consumer will fetch the next message.
2. Committing an offset greater than the current offset would indicate acknowledging messages that have not yet been consumed, which is not logical or allowed.

The committed offset is always less than or equal to the current offset.

---

## Different Types of Acknowledgments While Committing Messages

Kafka provides different strategies for acknowledgment when committing messages, which dictate how offsets are managed and when they are committed:

### 1. Automatic Commit (`enable.auto.commit=true`):

- The consumer automatically commits the offset periodically based on the configured interval (`auto.commit.interval.ms`).
- **Pros:**
  - Simplifies implementation.
- **Cons:**
  - Can lead to message loss or duplication if the consumer fails between polls and commits.

### 2. Manual Commit (Synchronous):

- The consumer explicitly calls `commitSync()` to commit the offsets after processing messages.
- **Pros:**
  - Ensures offsets are committed only after successful processing.
- **Cons:**
  - Blocking operation; can introduce latency.

### 3. Manual Commit (Asynchronous):

- The consumer calls `commitAsync()` to commit offsets without blocking.
- **Pros:**
  - Non-blocking; faster.

- **Cons:**
  - May lose commit information if the consumer crashes before the callback completes.

#### 4. Commit with Transactional Processing:

- In a Kafka Streams application or with Kafka's transactions, offset commits can be tied to a transaction.
- This ensures **exactly-once processing** semantics.
- **Pros:**
  - Guarantees consistency between data processing and offset commits.
- **Cons:**
  - More complex and higher latency.

#### 5. Custom Acknowledgment Mechanism:

- Developers can implement custom acknowledgment mechanisms to manage offsets, based on specific use cases.

---

## Summary

- The current offset represents the next unread message, while the committed offset tracks the last successfully processed and acknowledged message.
- The committed offset cannot exceed the current offset.
- Kafka provides several acknowledgment mechanisms (``auto``, ``manual-sync``, ``manual-async``, and transactional) to balance fault tolerance, performance, and processing guarantees.