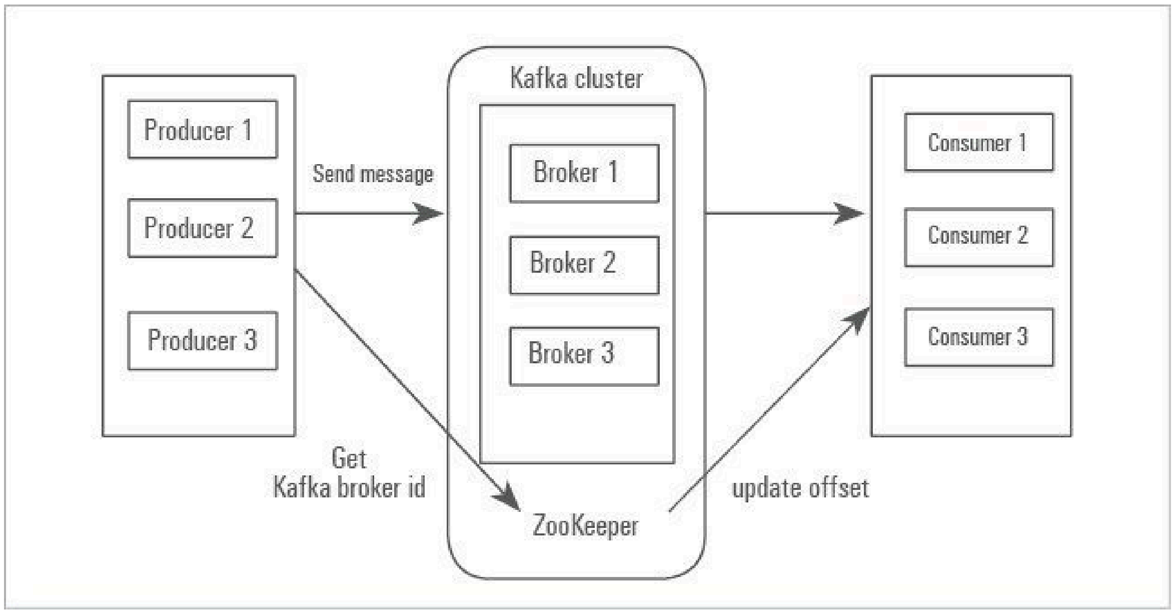
What is Apache Kafka?

1. Open source distributed streaming platform for processing large volumes of streaming data from real-time applications.
2. Designed for high-throughput, fault-tolerant, and real-time data streaming and messaging purpose.
3. Widely used for building data pipelines, event-driven architectures, and real-time analytics.

What is the difference between Kafka and traditional message brokers like JMS?

Apache Kafka	JMS (IBM MQ, Rabbit MQ)
Data Streaming Primarily designed for data streaming and real-time event processing. It excels at handling large volumes of data and provides a distributed log-based architecture for event-driven applications.	**Point to Point Messaging** JMS are designed for point-to-point messaging and publish-subscribe messaging patterns. They are typically used for reliable messaging between applications and systems.
Message Persistence Kafka retains messages for a configurable period, allowing consumers to rewind and replay messages. Often used as a data store for event sourcing and stream processing. Kafka's message storage is log-based, providing strong durability guarantees.	**Message Persistence** JMS usually persist messages for a shorter duration and are primarily focused on ensuring reliable message delivery between sender and receiver.
Scalability Kafka is horizontally scalable and designed for handling high throughput, making it well-suited for scenarios with massive data streams and real-time analytics.	**Scalability** Have some degree of scalability, but they may require more complex setups to achieve the same level of scalability as Kafka.
Use Cases Log aggregation, data pipelines, event sourcing, real-time analytics, and building event-driven architectures.	**Use Cases** Used for reliable communication between enterprise applications, especially in scenarios where transactional messaging and guaranteed delivery are critical.
Partitioning, Data Replication, Message Order Guarantee	

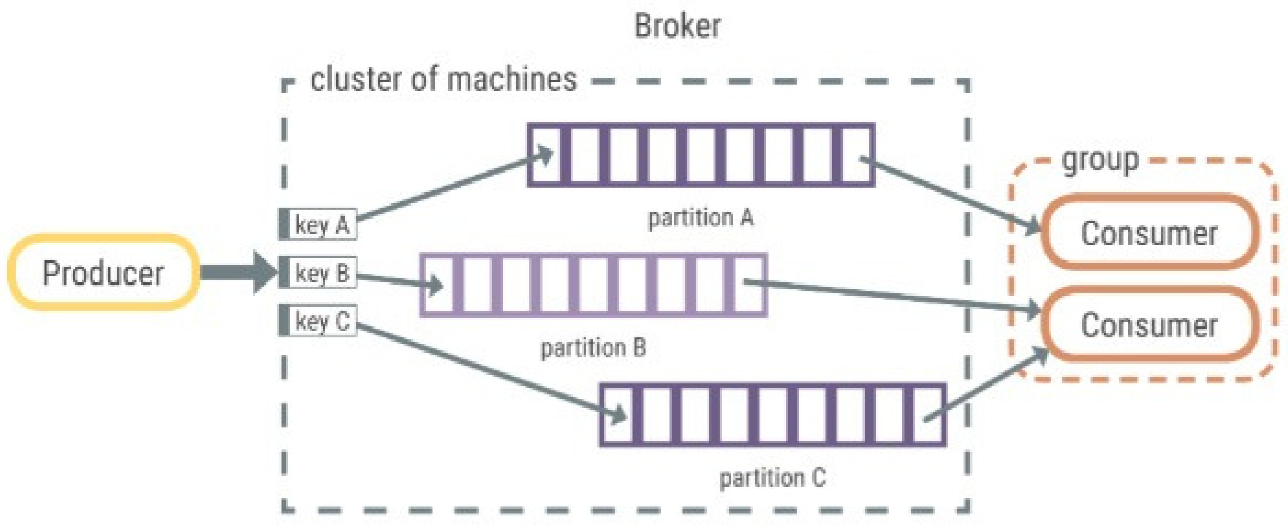
What are the key components in Apache Kafka? (Explain Architecture of Apache Kafka)



Components	Description
Brokers	▪ Kafka Server is known as broker. Its bridge between producers and consumers. ▪ Broker receives data from producers, store it in topics, and serve it to consumers. ▪ It also manages topics & partitions. Kafka can have multiple brokers for scalability and fault tolerance.
Zookeeper	▪ A centralized service for maintaining Kafka cluster metadata. ▪ Kafka has moved away from direct dependence on Zookeeper for newer versions. In newer versions, Kafka has its own internal metadata management.
Topic	▪ A topic is a logical channel of stream of data. Producer publishes data to the topic and same data is consumed by consumer. ▪ Topics are divided into partitions to enable parallel processing and distribution of data across brokers.
Producer	▪ Clients that publish data to Kafka topics. Ex. Java/Spring Boot App, Source connectors etc.
Consumer	▪ Clients that subscribe to topics and process the data. Ex. Java/Spring Boot App Kafka listener, Sink Connecters etc.

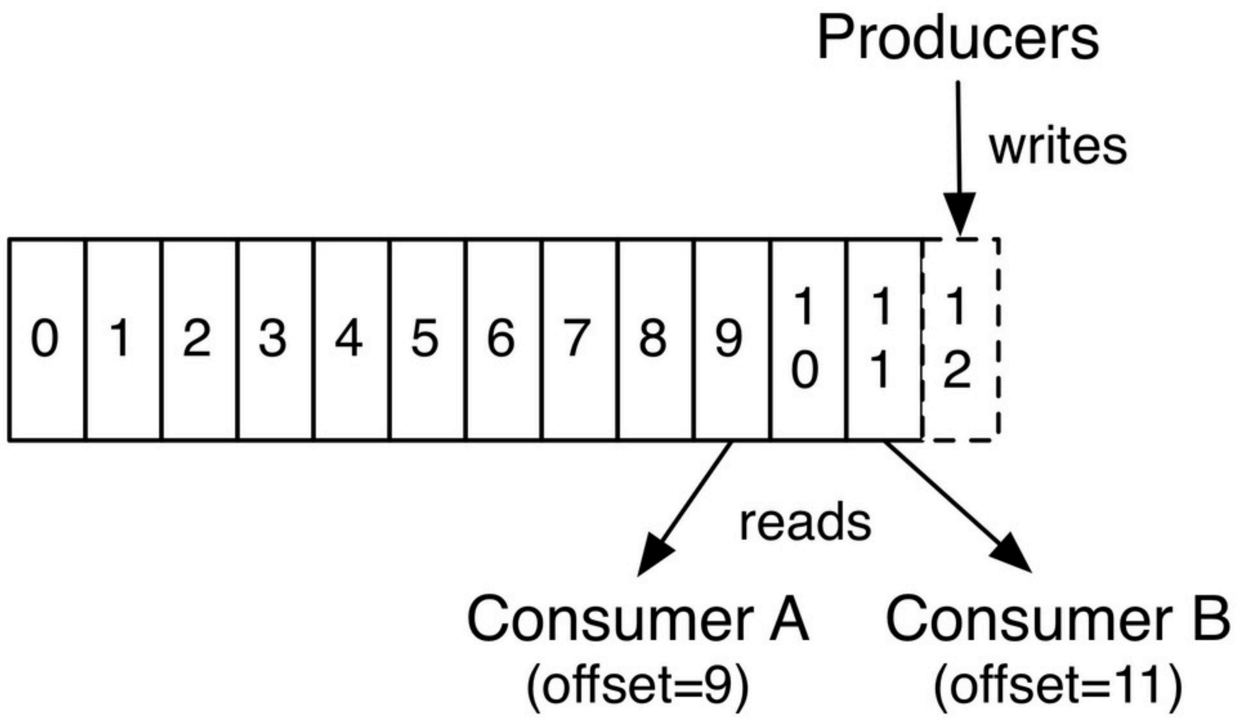
What is Kafka Partition? What's the purpose of it?

Partitioning takes the single topic and breaks it into multiple fractions (partitions), each of which can live on a separate node or broker in the Kafka cluster. That number of fractions is determined by us or by the cluster default configurations. This way, the work of storing messages, writing new messages, and processing existing messages can be split among many nodes in the cluster. Kafka guarantees the order of the events within the same topic partition. However, by default, it does not guarantee the order of events across all partitions. Purpose of partitioning is to enable parallelism and increase throughput.



What is Kafka Offset? What is Kafka Commit?

Kafka Offset	Kafka Commit
▪ Unique identifier or pointer of message/record within partition of topic. It represents position of the consumer in the partition. Ex. Current offset of consumer is 5 means it has consumed records from offset 0 to 4 and will receive next record with offset 5. ▪ Offsets are unique within a partition but not across partitions or topics. ▪ Each consumer maintains its own offset for each partition it is consuming from. This means that different consumers can be at different positions within the same partition, allowing for independent and parallel processing of messages.	▪ Act of confirming that a consumer has successfully processed and consumed a batch of messages up to a certain offset within a partition. ▪ Ex. Current offset of consumer A is 5. It can be confirmed only when Kafka performs commit operation to inform Zookeeper that consumer A has successfully received records from offset 0 to 4. ▪ Committing offsets is important for ensuring that in case of a consumer failure or a system crash, the consumer can resume from where it left off and not reprocess the same messages. ▪ Kafka provides two main approaches for committing offsets Manual Commit & Auto Commit.



What is the difference between Manual Commit & Auto Commit?

Manual Commit	Auto Commit
▪ Manual offset commit can be enabled by setting **enable.auto.commit** to false. ▪ Here consumer explicitly commits offsets using the commitSync or commitAsync methods provided by the Kafka client library. ▪ **commitSync** is a synchronous commit and will block the thread until either the commit succeeds, or an unrecoverable error is encountered. ▪ **commitAsync** is asynchronous commit operation. It won't block the thread. If any error encountered, then it passed to callback method.	▪ Consumer periodically commits offsets automatically based on a configured interval. Default interval is 5 sec. ▪ This is the simplest way to commit offsets by just setting **enable.auto.commit** property to true at Kafka consumer side. We can also change default commit interval by setting **auto.commit.interval.ms** to specific interval. ▪ Sometimes auto commit could also lead to duplicate processing of records in case consumer crashes before the next auto commit interval.

Explain the Kafka log retention mechanism and its configuration

- Retention period refers to the length of time that messages are kept in a Kafka topic before they are discarded. By default, Kafka retains data for seven days.
- This is specified by the log.retention.hours parameter in the broker configuration, which is set to 168 hours (7 days * 24 hours). Retention period can be adjusted according to the needs of your application.
- Kafka supports a server-level retention policy that can be tuned by configuring exactly one of the three time-based configuration properties: log.retention.hours, log.retention.minutes, log.retention.ms.
- Kafka overrides a lower-precision value with a higher one. So, log.retention.ms would take the highest precedence.
- If the log retention is set to five days, then the published message is available for consumption five days after it is published. After that time, the message will be discarded to free up space.
- The performance of Kafka is not affected by the data size of messages, so retaining lots of data is not a problem.

Explain the role of a Kafka Producer & Consumer

Kafka Producer Responsible for publishing data streams to one or more topics in the Kafka cluster. Producers can publish data from various sources, such as applications, databases, sensors, or file systems etc.

- **Data Serialization:** Producers serialize the data into a format suitable for transmission over the network and storage in Kafka.
- **Load Balancing:** Producers automatically distribute data across the partitions of a topic based on record key, ensuring efficient parallelization and scalability.
- **Durability & Availability:** Producers handle scenarios where Kafka brokers are unavailable or fail during message publication, ensuring data durability and guaranteed delivery.

Kafka Consumer Responsible for reading data streams from one or more topics in the Kafka cluster.

- **Data Deserialization:** Consumers deserialize the data received from Kafka brokers into a format that can be processed by the consumer application. Ex. Avro
- **Offset Management:** Consumers keep track of their position in the log (offset) for each partition they are consuming from, allowing them to resume consumption from the last committed offset in case of failures or restarts.
- **Group Management:** Consumers can be part of a consumer group to distribute partitions among themselves.
- **Data Processing:** Processing the data received from Kafka, applying transformations, filtering, aggregations, or integrating it with other systems or applications.

Producer Example

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class KafkaProducerExample {
    public static void main(String[] args) {
        // Configure Kafka producer properties Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); props.put("key.serializer",
        "org.apache.kafka.common.serialization.StringSerializer"); props.put("value.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");


        // Create a Kafka producer instance KafkaProducer<String, String>
        producer = new KafkaProducer<>(props);

        // Publish messages to the "my-topic" topic
        for (int i = 0; i < 10; i++) {
            String message = "Message " + i; ProducerRecord<String, String> record = new
            ProducerRecord<>("my-topic", message); producer.send(record);

        }

        // Flush and close the producer
        producer.flush();
        producer.close();
    }
}
```

Consumer Example

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {
    public static void main(String[] args) {
        // Configure Kafka consumer properties
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "my-consumer-group");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        // Create a Kafka consumer instance
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // Subscribe to the "my-topic" topic
        consumer.subscribe(Collections.singletonList("my-topic"));

        // Poll and consume records
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.println("Key: " + record.key() + ", Value: " + record.value());
            }
        }
    }
}
```

What is a Kafka producer callback, and why would you use it?

Kafka producer callback is a method that you can implement to handle responses from the broker asynchronously. This is particularly useful because Kafka producers are asynchronous by nature. When you send a message using a Kafka producer, the send() method adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.

However, you might want to know more about the outcome of your message production, such as whether the data was correctly produced, where it was produced, its offset and partition value, etc. This is where callbacks come in handy. We can overload the KafkaProducer.send method with an instance of the Callback interface as the second parameter.

Producer Callback

```
producer.send(record, new Callback() {
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                if (exception == null) {
                    System.out.println("Message sent successfully to topic: " + metadata.topic() +
                            ", partition: " + metadata.partition() +
                            ", offset: " + metadata.offset());
                } else {
                    System.err.println("Error sending message: " + exception.getMessage());
                }
            }
        });
```

Explain Kafka Producer Acknowledgment

In Apache Kafka, a producer can choose to receive an acknowledgment of data writes. This acknowledgment is also known as a confirmation1. There are three acknowledgment modes available.

- **acks=0 (possible data loss):** The producer just sends the data and does not wait for an acknowledgment. This can possible data loss because if the broker is down, the producer will not know about it because it will not get an acknowledgment.
- **acks=1 (limited data loss):** This is the default setting as of Kafka 2.01. Producer requests response from leader and leaving replicas. If an acknowledgement is not received from leader, then producer has to retry.
- **acks=all (no data loss):** Here producer requires acknowledgement from both leader and replicas. if it does not get response from leader & all replicas, then producer will retry publishing message.

How can you scale a Kafka consumer to handle higher loads?

There are several ways to scale a Kafka consumer to handle higher loads:

- **Partitioning:** Kafka topics are divided into partitions, and each partition can be consumed by only one consumer within a consumer group. You can increase the number of partitions for the topics being consumed, which allows for more parallelism in processing messages.
- **Increasing Consumer Instances:** As you have increased number of partitions on topic, same way you can now increase number of consumer instances within consumer group in order to leverage full benefit of parallelism. Kafka ensures that each partition is consumed by only one consumer within consumer group, so adding more instances will distribute the load across them.
- **Tuning Consumer Configurations:** Adjusting consumer configurations can also improve the scalability and performance of Kafka consumers. Parameters such as fetch.min.bytes, fetch.max.bytes, max.poll.records, and max.poll.interval.ms can be tweaked to optimize the balance between throughput and resource utilization.
- **Optimizing Message Processing:** Ensure that the message processing logic in your consumer application is efficient. This includes minimizing processing time, avoiding unnecessary operations, and using asynchronous processing where possible.

What is a Kafka consumer lag, and why is it important to monitor?

Kafka consumer lag refers to the difference between the offset of latest message inserted in a Kafka partition and the offset of the last message that has been consumed by a consumer. It represents the number of messages that the consumer is yet to process.

- **Identifying Bottlenecks:** High consumer lag can indicate that the consumer is struggling to keep up with the rate of incoming messages. This could be due to issues with the consumer application itself, resource constraints (CPU, memory, network).
- **Preventing Data Loss:** If a consumer falls too far behind and the retention period for the topic expires, there is a risk of data loss. Messages removed as part of retention will no longer be processed.
- **Ensure Timely Processing:** For applications that require near real-time processing of data, high consumer lag can lead to increased latency and delayed response times, potentially impacting the system's overall performance and user experience.
- **Load Balancing:** In a consumer group, monitoring the lag for individual consumer instances can help identify if the load is evenly distributed or if some consumers are falling behind, indicating the need for rebalancing or scaling the group.
- **Capacity Planning:** Tracking consumer lag over time can provide insights into message throughput patterns and help plan for future capacity requirements, such as scaling the number of consumer instances or adjusting the hardware resources allocated to the consumer application.
- **Alerting & Troubleshooting:** By setting appropriate thresholds for consumer lag, alerts can be triggered when the lag exceeds a certain level, allowing for proactive monitoring and investigation of potential issues.

What happens if consumer fails to commit offset?

If consumer fails to commit offset, then it leads to several potential issues:

- **Duplicate Message Processing:** When consumer fails to commit offsets, it may start reading the messages from topic from the last committed offset. This can result in duplicate message processing by consumer, leading to data inconsistency or incorrect application behavior.
- **Loss of Message Ordering:** Kafka guarantees message ordering within a partition, but if a consumer fails to commit offsets and reprocesses messages, it may disrupt the order of messages consumed by downstream applications.
- **Increased Consumer Lag:** If offsets are not committed in a timely manner, the consumer will fall behind the latest offsets in the partitions, leading to increased consumer lag.
- **Inconsistent Consumer Group State:** Failed offset commits can result in inconsistent consumer group states, where some consumers have committed offsets while others have not. This can lead to rebalancing issues and uneven distribution of partitions among consumers within the group.

Explain the Kafka replication protocol and its guarantees

The Kafka replication protocol is responsible for maintaining multiple copies (replicas) of partitions across brokers in a Kafka cluster. It provides several guarantees to ensure data durability, consistency, and fault tolerance.

- **Leader-Follower Replication Model:** Kafka uses a leader-follower replication model, where each partition in a topic has one leader and zero or more followers. The leader is responsible for handling all read and write requests for the partition, while followers replicate the leader's data asynchronously.
- **Replica Synchronization:** When a leader receives a produce request (i.e., a message to be appended to the partition), it first appends the message to its local log and then asynchronously replicates the message to its followers. Followers pull data from the leader using the fetch request protocol. The leader sends batches of messages to followers, and followers acknowledge messages once they are successfully replicated.
- **In-Sync Replicas (ISR):** Kafka maintains a set of in-sync replicas (ISR) for each partition. These are the replicas that are fully caught up with the leader's log. The ISR ensures that a partition can continue to function even if some replicas fall behind or become unavailable.
- **Leader Election:** If the leader of a partition becomes unavailable (e.g., due to node failure), Kafka triggers a leader election process to select a new leader from the ISR. This ensures that data availability is not compromised even in the event of leader failures.
- **Durability and Consistency Guarantees:** Kafka provides strong durability and consistency guarantees. Once a message is successfully written to the leader's log and replicated to all ISR replicas, it is considered durable and will not be lost even in the event of node failures.

What are the different delivery semantics in Kafka?

Kafka supports three different delivery semantics for message processing: at-most-once, at-least-once, and exactly-once. These delivery semantics determine how messages are produced and consumed within Kafka.

- **At Most Once Delivery:** In this semantic, a message produced by producer is either delivered and processed once or not at all. Producers do not wait for acknowledgment from Kafka brokers, hence there is no guarantee that the message will be processed, as it may be lost or discarded due to various reasons, such as broker failures or network issues. This mode is suitable for scenarios where message loss is acceptable, and low latency is crucial.
- **At Least Once Delivery:** Kafka guarantees that a message will be delivered and processed at least once. Here producer waits for acknowledgment (ACK) from Kafka brokers after sending message, ff a message is not acknowledged within a specified timeout period, the producer retries sending the message. It is possible that the message may be processed multiple times due to retries or reprocessing. At-least-once delivery is suitable for use cases where duplicate processing is acceptable or can be handled by making the consumer processing logic idempotent.
- **Exactly Once Delivery:** This is the strongest delivery semantic, where Kafka ensures that each message is delivered and processed exactly once, even in the presence of failures. This mode is achieved through Kafka's transactional support, which ensures that both message production and consumption are part of the same atomic transaction. Producers use transactions to write messages to Kafka topics atomically, ensuring that messages are either fully committed or fully rolled back in case of failure. Consumers use idempotent processing and transactional offsets to guarantee exactly once processing. This semantic is suitable for use cases that require strict data consistency and cannot tolerate duplicate processing or data loss, such as financial transactions.

What are Kafka Streams?

Kafka Streams is a client library for building real-time, fault-tolerant, and scalable stream processing applications on top of Apache Kafka. It is part of the Apache Kafka project and provides a lightweight and easy-to-use API for performing stream processing tasks such as data transformation, aggregation, filtering, and joining.

Key features of Kafka Streams include:

- **Stateful Stream Processing:** Kafka Streams provides built-in support for stateful stream processing, allowing applications to maintain and query state as they process data streams. State can be stored locally within the application or backed by external state stores such as Apache Kafka's Streams State Store or external databases like RocksDB. Ex. Windowing, Aggregations etc.
- **Exactly Once Semantics:** Offers support for exactly once semantics, ensuring that data processing is idempotent and resilient to failures.
- **High-Level DSL & Processor API:** Kafka Streams provides both a high-level DSL (Domain-Specific Language) and a lower-level Processor API for defining stream processing topologies. The DSL simplifies the development of common stream processing tasks with intuitive constructs like map, filter, groupByKey, aggregate, and join. The Processor API offers more flexibility for custom stream processing logic by allowing developers to define their own processor nodes within the processing topology.
- **Fault Tolerance and Scalability:** Inherently fault-tolerant and scalable, leveraging Kafka's distributed nature and built-in replication mechanisms.
-

How does Kafka handle data security and encryption?

Kafka provides several mechanisms to handle data security and encryption:

- **Encryption in Transit:** Supports SSL/TLS encryption for communication between clients and brokers, as well as communication between brokers in a cluster. By enabling SSL/TLS, all data transmitted over the network is encrypted, protecting it from unauthorized access. Client authentication can be configured using SSL/TLS client certificates or other mechanisms like SASL (Simple Authentication and Security Layer).
- **Encryption at Rest:** Kafka supports transparent data encryption at rest, meaning that data stored on disk (in log segments) is automatically encrypted. Encryption at rest protects data from unauthorized access in case of disk theft or unintended exposure.
- **Authentication & Authorization:** Provides various mechanisms for client authentication, including SSL/TLS client certificates, SASL etc. Once authenticated, clients can be authorized to perform specific actions (read, write, etc.) on topics and clusters using Access Control Lists (ACLs).
- **Data Integrity:** Ensures data integrity by using checksums for messages stored on disk. This helps detect and prevent data corruption or tampering.