**1.What is Kafka?**

**Kafka** is an **open-source messaging system** that was created by LinkedIn and later donated to the Apache Software Foundation. It's built to handle large amounts of data in real time, making it perfect for creating systems that respond to events as they happen.

Kafka organizes data into categories called "topics." Producers (apps that send data) put messages into these topics, and consumers (apps that read data) receive them. Kafka ensures that the system is reliable and can keep working even if some parts fail.

Event Streaming :

1. Creating Real time stream
2. Processing Real time stream

Paytm is API Paytm method lets assume, I am using Paytm to do some payments for I am booking movie tickets Paytm provide feature to do 10 types of transactions. Now when I am doing transactions that request go to Kafka sever but I am not the Paytm user who is doing transactions at this time.

**Creating Real time stream :**

This Kafka server receiving request million/Billions of requests each second/milli second. So, sending the stream of continues data Paytm to Kafka server called Creating Real time stream.

**Processing Real time stream:**

Paytm want’s restrict mark 10 transactions for a day means a user can only do 10 transactions in a day using Paytm method. If it is exceeding the limit then client application want to send a notification to the user in such a way that my client application need to continuously fetch the data and need to do the validations check to transaction count for specific user each every second/milli second this continuously listen to the Kafka message and process the it is called Processing Real time stream.

**Distributed :**

Distribute Multiple computers to different node/region to balance the load to avoid the downtime .

**2. Why we need Kafka ?**

I have two applications like application1 and application2. Now application1 want’s to send data to application2 but application 2 not available to receive the data then again loss the data to overcome this communication failure we might need to install something between two applications that comes in to picture we can add messaging system between two applications.

Here application1 send the data to Kafka if application2 is not available then applications2 collect the data from Kafka whenever come online that is way won’t loss the data.

**3.How does it work high level overview :**

Publisher is person who will publish the message/event to the messaging system(Kafka) and message will go and sit in message broker. Now subscriber will go to the message broker and will ask for the message broker and subscribe simply listen to message broker to get the messages.

**4.Core Components of Apache Kafka**

**1. Producers :** Its an application or service that **sends messages to a Kafka topic**. These processes **push data** into the Kafka system.

**2.Consumer :** Consumer act as a receiver is responsible for receiving / consuming a message.

**Applications that read data from Kafka.**

**3.Broker :** The Kafka broker is nothing but just a server. In simple term, A broker is an intermediate entity that helps in message exchange between a producer and consumer.

**4.Cluster :** Cluster is common terminology in the distributed system is nothing but group of computers or servers that we are working for common purposes.

Kafka is also distributed system it can have multiple Kafka servers/brokers inside single Kafka cluster. There can be one more broker in the Kafka cluster.

**5. Kafka Topic**

A topic in Kafka is a category or feed name to which messages are published.

Kafka messages are always associated with topics, and when you want to send a message, you send it to a specific topic.

**6.Partitions :**

Kafka is distributed system we can break the Kafka into multiple parts and distribute parts into the different messages this concept is called partition.

**7.Offset :**

Producers send a message it will go and sit in any partition inside a topic it will work Round Robin principle as soon as message arrive in partition number into that message that is called offset.

Or

In Kafka a sequence number is assigned to each message in each partition of a Kafka topic this sequence number is called Offset.

**8. Consumer Groups**

Kafka allows consumer groups, where multiple consumers can read message from the same topic, but Kafka ensures that each message is processed by only one consumer in the group.

This helps with load balancing and allows consumers to read messages starting from any offset.

**5. Zookeeper**

Kafka uses**Apache Zookeeper** to manage metadata, control access to Kafka resources**.** Zookeeper ensures high availability by making sure the Kafka cluster remains functional even if a broker fails.

**Kafka Components**

* **Topics**: Categories where messages are published
* **Producers**: Applications that send data to Kafka
* **Consumers**: Applications that read data from Kafka
* **Brokers**: Servers that store and manage the data
* **Partitions**: Subdivisions of topics that allow parallel processing

**5.How Apache Kafka Works**

Apache Kafka moves data from one place to another in a smooth and reliable way. Here’s how it works in simple terms:

**Step 1: Producers Send Data**

* Producers are applications that **create data** and send it to Kafka.
* This data can be anything—logs, transactions, user activities, or events.
* Kafka **splits** the data into smaller parts called **partitions**, making it easier to handle large amounts of information.

**Step 2: Kafka Stores the Data**

* Kafka **organizes** the data into **topics**, where it is saved for a certain period.
* Even if a consumer reads the data, Kafka **doesn’t delete it immediately**.
* To prevent data loss, Kafka **makes copies** of the data and stores them on different servers.

**Step 3: Consumers Read the Data**

* Consumers are applications that **subscribe to topics** and read messages.
* To manage the load, consumers are divided into **consumer groups**, so no message is processed twice.
* Consumers can **choose where to start reading**, whether from the newest message or an earlier point.

**Step 4: Kafka Balances the Load**

* **ZooKeeper** helps Kafka manage which server is in charge of storing and distributing data.
* If a server goes down, Kafka automatically **redirects the data** to another server.

**Step 5: Data is Processed and Used**

* Once consumers receive the data, they can **store it in a database, analyze it, or trigger other events**.
* Kafka can work with tools like **Apache Spark, Flink, and Hadoop** for deeper analysis.

**6.How Kafka Integrates Different Data Processing Models**

Apache Kafka is highly versatile and can seamlessly integrate various data processing models, including **event streaming**, **message queuing**, and **batch processing**.

**1. Event Streaming (Publish-Subscribe Model)**

Kafka’s primary function is **event streaming**, where:

* **Producers** (applications sending data) publish messages to Kafka topics.
* **Consumers** (applications reading data) subscribe to topics and receive messages as soon as they arrive.
* Multiple consumers can read the same message, allowing for real-time data distribution.

***Example****: A stock trading platform can use Kafka to stream live market data to multiple dashboards.*

**2. Message Queue (Point-to-Point Processing)**

Kafka can also act like a **message queue** by using **consumer groups**:

* When multiple consumers are in the same group, Kafka **distributes messages among them**, ensuring each message is processed only once.
* This setup helps in **load balancing**, making sure no single consumer is overwhelmed.

***Example****: A ride-hailing app like Uber can use Kafka to assign incoming ride requests to available drivers efficiently.*

**3. Batch Processing**

Even though Kafka is designed for real-time data, it can also handle **batch processing**:

* Messages can be stored in Kafka topics and processed later.
* Tools like **Apache Spark or Hadoop** can read data from Kafka in batches and perform analytics.

***Example****: An e-commerce company can collect website visitor data in Kafka and analyze it later to improve product recommendations.*

**4. Hybrid Model (Real-Time + Batch Processing)**

Kafka is flexible enough to support a **mix of real-time and batch processing**:

* It can send data immediately for real-time analytics while also storing it for batch processing later.
* This is often done using **Kafka Streams, Spark Streaming, or Flink**.

**Example**: A fraud detection system can process transactions in real time to flag suspicious activity while also running deeper batch analysis at the end of the day.

Interview Questions

<https://www.geeksforgeeks.org/apache-kafka/kafka-interview-questions/>

**7.What is Kafka Schema ?**

**Kafka schema** defines the structure of the data being transmitted between producers and consumers in Apache Kafka. Think of it as a contract that ensures everyone in the system agrees on what a message should look like.

**Why Kafka Schemas Matter**

* **Data consistency**: Ensures that messages have the expected fields and types.
* **Validation**: Prevents malformed or incomplete data from being processed.
* **Interoperability**: Allows different applications to communicate reliably.
* **Documentation**: Acts as a blueprint for understanding the data model.

**Common Schema Formats in Kafka**

Kafka itself doesn’t enforce schemas, but tools like **Confluent Schema Registry** enable schema management using formats like:

* **Avro**: Compact and fast, ideal for Kafka.
* **Protobuf**: Efficient and extensible, used in many modern systems.
* **JSON Schema**: Human-readable but less compact.

Schemas can evolve over time. With proper compatibility settings (like backward or forward compatibility), older consumers can still read newer messages and vice versa.

**8.Vertical scaling and Horizontal scaling in Kafka ?**

In Apache Kafka, **scaling** is essential to handle increasing data loads and ensure high availability. There are two primary strategies: **vertical scaling** and **horizontal scaling**, each with its own trade-offs.

**Vertical Scaling (Scaling Up)**

This means upgrading the **hardware resources** (CPU, RAM, disk) of existing Kafka brokers.

Pros:

* Simpler to manage fewer nodes.
* Lower network overhead between brokers.
* Easier to configure and monitor.

Cons:

* Limited by the maximum capacity of a single machine.
* Hardware upgrades can be expensive.
* A single point of failure becomes more critical.

**Horizontal Scaling (Scaling Out)**

This involves **adding more Kafka brokers** (nodes) to the cluster.

Pros:

* Virtually unlimited scalability.
* Better fault tolerance and high availability.
* Load can be distributed across more machines.

Cons:

* More complex to manage and monitor.
* Requires careful partition and replication planning.
* Network traffic between brokers increases

**9.Retry mechanism in Kafka ?**

Apache Kafka provides robust **retry mechanisms** to ensure reliable message delivery in the face of transient failures. These mechanisms apply to both **producers** and **consumers**, and can be customized depending on the use case.

**Kafka Producer Retry Mechanism**

When a producer fails to send a message (e.g., due to network issues or broker unavailability), Kafka can automatically retry.

Key Configurations:

* retries: Number of retry attempts (default is 2147483647 in newer versions).
* retry.backoff.ms: Time to wait between retries.
* acks: Controls acknowledgment behavior (all ensures the highest durability).
* delivery.timeout.ms: Total time to deliver a message, including retries.

This retry logic is **synchronous** and happens before the message is considered failed.

**Kafka Consumer Retry Mechanism**

Consumers may fail to process a message due to business logic errors or downstream service issues. Kafka itself doesn’t provide built-in consumer retries, but you can implement them using:

1. **Manual Retry Logic**

* Catch exceptions in the consumer and retry processing.
* Use exponential backoff or fixed delays.

2. **Dead Letter Topics (DLT)**

* Failed messages are redirected to a separate topic for later inspection or reprocessing.

3. **Spring Kafka Retry Support**

If you're using Spring Kafka:

* Use @Retryable and @Recover annotations.
* Configure retry templates with exponential backoff and max attempts.

4. **KafkaJS Retry Strategy (for Node.js)**

KafkaJS uses an exponential backoff strategy with jitter:

* Retry intervals grow with each attempt.
* Randomization prevents retry storms.

**10.How does data retention policy ?**

Kafka's **data retention policy** determines how long messages are stored in a topic before being deleted. This is crucial for managing disk usage and ensuring that consumers can access data for a defined period.

Types of Retention Policies

1. **Time-Based Retention**

* Messages are retained for a specified duration (e.g., 7 days).
* Controlled by the retention.ms setting.
* After the time expires, Kafka marks the data for deletion.
* Example: retention.ms=604800000 (7 days)

2. **Size-Based Retention**

* Messages are retained until the log size exceeds a configured threshold.
* Controlled by retention.bytes.
* Useful when disk space is a constraint.

3. **Log Compaction**

* Retains only the latest value for each key.
* Ideal for changelog-style topics.
* Controlled by cleanup.policy=compact.

**11.What is purpose of idempotent producer in Kafka ?**

The purpose of an idempotent producer in Apache Kafka is to ensure exactly-once delivery semantics by preventing duplicate messages during retries.

**Why Idempotent Producers Matter**

In distributed systems, failures can happen—network glitches, broker restarts, or timeouts. When a Kafka producer retries sending a message, it might accidentally send the same message more than once. This leads to duplicate records, which can corrupt downstream processing.

What Idempotent Producers Do

* Assign a **unique sequence number** to each message.
* Kafka brokers use this sequence to detect and discard duplicates.
* Guarantees **exactly-once delivery** per partition (not across partitions or topics).

**How to Enable It**

Just set:

**enable.idempotence=true**

This automatically configures:

* acks=all
* retries to a high number
* max.in.flight.requests.per.connection=5 (to preserve order)