 𝗞̲𝗮̲𝗳̲𝗸̲𝗮̲ ̲𝗦̲𝗲̲𝗿̲𝗶̲𝗲̲𝘀̲ ̲(̲𝗣̲𝗮̲𝗿̲𝘁̲ ̲𝟭̲)̲  
  
Before understanding Kafka, let's first understand a few terminologies.  
  
✅ 𝙋𝙪𝙗𝙡𝙞𝙨𝙝-𝙎𝙪𝙗𝙨𝙘𝙧𝙞𝙗𝙚:  
The publish-subscribe model is a messaging pattern used in distributed systems, where:  
✔ Senders of messages (publishers) do not send messages directly to specific receivers (subscribers).  
✔ Publishers send messages to an intermediary called a 𝗠𝗲𝘀𝘀𝗮𝗴𝗲 𝗕𝗿𝗼𝗸𝗲𝗿 which then distributes the messages to all interested subscribers.  
  
In this model:  
✔ Publishers and subscribers are 𝗱𝗲𝗰𝗼𝘂𝗽𝗹𝗲𝗱 𝗳𝗿𝗼𝗺 𝗲𝗮𝗰𝗵 𝗼𝘁𝗵𝗲𝗿, i.e., that they do not need to know about the existence of each other.  
✔ Publishers only need to know the topic or channel to which they want to publish messages, and subscribers only need to know the topics or channels to which they want to subscribe.  
  
✅ 𝙈𝙚𝙨𝙨𝙖𝙜𝙚 𝘽𝙧𝙤𝙠𝙚𝙧  
✔ It is an intermediary service that enables communication between applications or components by transmitting messages between them.  
✔ It typically provides a set of features, including message routing, transformation, and filtering, as well as scalability, reliability, and fault-tolerance.  
✔ It is commonly used in enterprise integration and distributed systems, including microservices architectures, event-driven systems, and service-oriented architectures.  
✔ Some of the popular message brokers are RabbitMQ, IBM MQ, Apache Kafka, Amazon SQS, Apache Pulsar, etc.  
  
✅ 𝙀𝙫𝙚𝙣𝙩 𝙎𝙩𝙧𝙚𝙖𝙢𝙞𝙣𝙜  
Event streaming is the practice of:  
✔ Capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events.  
✔ Storing these event streams durably for later retrieval.  
✔ Manipulating, processing, and reacting to the event streams in real-time.  
✔ Routing the event streams to different destination technologies as needed.  
  
✅ 𝙒𝙝𝙖𝙩 𝙚𝙭𝙖𝙘𝙩𝙡𝙮 𝙞𝙨 𝙆𝙖𝙛𝙠𝙖?  
📌 It is an open-source distributed 𝗘𝘃𝗲𝗻𝘁 𝗦𝘁𝗿𝗲𝗮𝗺𝗶𝗻𝗴 platform.  
📌 It is used for building real-time data pipelines and streaming applications.  
📌 It offers a 𝗣𝘂𝗯𝗹𝗶𝘀𝗵-𝗦𝘂𝗯𝘀𝗰𝗿𝗶𝗯𝗲 based messaging system.  
📌 It was originally developed at 𝗟𝗶𝗻𝗸𝗲𝗱𝗜𝗻 as a stream processing platform and was subsequently open sourced in the late 2010 and accepted as an Apache Software Foundation incubator project in July 2011.  
📌 The development team at LinkedIn was led by 𝗝𝗮𝘆 𝗞𝗿𝗲𝗽𝘀 (𝗰𝘂𝗿𝗿𝗲𝗻𝘁𝗹𝘆, 𝗖𝗼-𝗳𝗼𝘂𝗻𝗱𝗲𝗿 𝗮𝗻𝗱 𝗖𝗘𝗢 𝗮𝘁 𝗖𝗼𝗻𝗳𝗹𝘂𝗲𝗻𝘁).

📌 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙖 𝙢𝙚𝙨𝙨𝙖𝙜𝙚?  
✔ It is the unit of data that is produced and consumed by Kafka clients.  
✔ It is simply an array of bytes, as far as Kafka is concerned.  
   
✅ Each message in Kafka has two parts: a 𝗸𝗲𝘆 and a 𝘃𝗮𝗹𝘂𝗲.  
✔ The key is used to determine the partition to which a message is assigned.  
✔ The value contains the actual data of the message.  
   
📌 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙖𝙣 𝙊𝙛𝙛𝙨𝙚𝙩?  
✔ It is another piece of metadata that Kafka adds to each message as it is produced.  
✔ It is an integer value that continually increases.  
✔ Each message in a given partition has a unique offset.  
   
📌 𝙒𝙝𝙖𝙩 𝙖𝙧𝙚 𝙏𝙤𝙥𝙞𝙘𝙨 𝙖𝙣𝙙 𝙋𝙖𝙧𝙩𝙞𝙩𝙞𝙤𝙣𝙨?  
✔ Messages in Kafka are categorized into topics.  
✔ You may think of a topic as a database table or a folder in a filesystem.  
✔ Kafka topics are broken down into a number of Partitions.  
   
✅ 𝗜𝗺𝗽𝗼𝗿𝘁𝗮𝗻𝘁:  
✔ There is 𝗻𝗼 𝗴𝘂𝗮𝗿𝗮𝗻𝘁𝗲𝗲 of message ordering 𝗮𝗰𝗿𝗼𝘀𝘀 𝘁𝗵𝗲 𝗲𝗻𝘁𝗶𝗿𝗲 𝘁𝗼𝗽𝗶𝗰 as the topic is broken down into several partitions.  
✔ There is a 𝗴𝘂𝗮𝗿𝗮𝗻𝘁𝗲𝗲 of message ordering within a single partition.  
   
📌 𝙒𝙝𝙖𝙩 𝙖𝙧𝙚 𝙋𝙧𝙤𝙙𝙪𝙘𝙚𝙧𝙨?  
The Producers:  
✔ Are client applications that 𝗽𝘂𝗯𝗹𝗶𝘀𝗵 𝗺𝗲𝘀𝘀𝗮𝗴𝗲𝘀 to a Kafka cluster.  
✔ Can be written in a variety of programming languages using Kafka client libraries.  
✔ Produce message for a specific Kafka topic (sometimes also for a specific partition).  
   
✅ 𝗜𝗺𝗽𝗼𝗿𝘁𝗮𝗻𝘁:  
✔ By default, the producer will balance messages over all partitions of a topic 𝗲𝘃𝗲𝗻𝗹𝘆.  
✔ In some cases, the producer can send messages to a 𝘀𝗽𝗲𝗰𝗶𝗳𝗶𝗰 𝗽𝗮𝗿𝘁𝗶𝘁𝗶𝗼𝗻. This is achieved using the message "key" and a partitioner that generates a hash of the key and map it to a specific partition. This ensures that all messages produced with a given key will get written to the same partition.  
✔ The producers could also use a custom partitioner for mapping messages to partitions.  
   
📌 𝙒𝙝𝙖𝙩 𝙖𝙧𝙚 𝘾𝙤𝙣𝙨𝙪𝙢𝙚𝙧𝙨?  
The Consumers:  
✔ Are client applications that 𝗿𝗲𝗮𝗱 𝗱𝗮𝘁𝗮 𝗳𝗿𝗼𝗺 𝗞𝗮𝗳𝗸𝗮 𝘁𝗼𝗽𝗶𝗰𝘀.  
✔ Subscribe to one or more topics and reads the messages in the order in which they were produced to each partition.  
✔ Keeps track of which messages it has already consumed by keeping track of the offset of messages.

✅ 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙖 𝘾𝙤𝙣𝙨𝙪𝙢𝙚𝙧 𝙂𝙧𝙤𝙪𝙥?  
📌A consumer group in Kafka is a group of one or more consumers that work together to consume and process the messages from Kafka topics.  
📌 The group ensures that each partition is only consumed by one member.  
  
𝗜𝗺𝗽𝗼𝗿𝘁𝗮𝗻𝘁:  
✔ When a topic receives new messages, Kafka distributes those messages across the partitions of the topic.  
✔ Each consumer group is assigned a set of partitions to consume from.  
✔ Each consumer in the group reads from a unique subset of those partitions.  
✔ If a single consumer fails, the remaining members of the group will reassign the partitions being consumed to take over for the missing member.  
✔ Consumer groups in Kafka are useful for scaling the processing of messages across multiple consumers.  
  
✅ 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙖 𝘽𝙧𝙤𝙠𝙚𝙧 𝙞𝙣 𝙆𝙖𝙛𝙠𝙖?  
📌 A single Kafka server is called a broker.  
📌 It receives messages from producers, assigns offsets to them, and writes the messages to storage on disk.  
📌 It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published.  
📌 It store data in the form of topics, which are partitioned and distributed across multiple brokers in a cluster.  
  
✅ 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙆𝙖𝙛𝙠𝙖 𝙘𝙡𝙪𝙨𝙩𝙚𝙧?  
📌 A cluster is a group of broker servers that work together to provide a distributed messaging system.  
📌 Kafka clusters are designed to provide high availability, fault tolerance, and scalability by distributing data across multiple nodes.  
📌 Within a cluster of brokers, one broker will also function as the cluster controller (elected automatically from the live members of the cluster).  
  
𝗜𝗺𝗽𝗼𝗿𝘁𝗮𝗻𝘁:  
✔ A partition is owned by a single broker in the cluster, and that broker is called the 𝗹𝗲𝗮𝗱𝗲𝗿 of the partition.  
✔ A replicated partition is assigned to additional brokers, called 𝗳𝗼𝗹𝗹𝗼𝘄𝗲𝗿𝘀 of the partition.  
✔ All producers must connect to the leader in order to publish messages.  
✔ Consumers may fetch from either the leader or one of the followers.  
  
✅ 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙧𝙚𝙩𝙚𝙣𝙩𝙞𝙤𝙣 𝙥𝙤𝙡𝙞𝙘𝙮?  
📌 It is defined as the durable storage of messages for some period of time.  
📌 Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the partition reaches a certain size in bytes (e.g., 1 GB).  
📌 Once these limits are reached, the messages are expired and deleted.

📌 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙖 𝙈𝙞𝙧𝙧𝙤𝙧𝙈𝙖𝙠𝙚𝙧?  
✔ It is a tool included in the Kafka project used for 𝗿𝗲𝗽𝗹𝗶𝗰𝗮𝘁𝗶𝗻𝗴 𝗱𝗮𝘁𝗮 𝘁𝗼 𝗼𝘁𝗵𝗲𝗿 𝗰𝗹𝘂𝘀𝘁𝗲𝗿𝘀.  
✔ It is simply a Kafka consumer and producer, linked together with a queue.  
✔ Messages are consumed from one Kafka cluster and produced to another.  
✔ It is quite useful when there is the need for multiple Kafka clusters in multiple datacenters (for segregation of types of data, isolation for security requirements, disaster recovery, etc.).  
  
📌 𝙒𝙝𝙖𝙩 𝙞𝙨 𝙕𝙤𝙤𝙠𝙚𝙚𝙥𝙚𝙧?  
✔ It is like a centralized service that manages cluster memberships, relevant configurations, and cluster registry services.  
✔ It acts a Kafka 𝗰𝗹𝘂𝘀𝘁𝗲𝗿 𝗰𝗼𝗼𝗿𝗱𝗶𝗻𝗮𝘁𝗼𝗿 that manages cluster membership of brokers, producers, and consumers participating in message transfers via Kafka.  
✔ It also helps in 𝗹𝗲𝗮𝗱𝗲𝗿 𝗲𝗹𝗲𝗰𝘁𝗶𝗼𝗻 for a Kafka topic.  
   
✅ 𝙎𝙪𝙢𝙢𝙖𝙧𝙮 𝙤𝙛 𝙆𝙖𝙛𝙠𝙖 𝙛𝙚𝙖𝙩𝙪𝙧𝙚𝙨:  
📌 𝗠𝘂𝗹𝘁𝗶𝗽𝗹𝗲 𝗣𝗿𝗼𝗱𝘂𝗰𝗲𝗿𝘀  
Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic.  
  
📌 𝗠𝘂𝗹𝘁𝗶𝗽𝗹𝗲 𝗖𝗼𝗻𝘀𝘂𝗺𝗲𝗿𝘀  
Kafka is designed for multiple consumers to read any single stream of messages without interfering with other clients.  
  
📌 𝗗𝗶𝘀𝗸-𝗕𝗮𝘀𝗲𝗱 𝗥𝗲𝘁𝗲𝗻𝘁𝗶𝗼𝗻  
Durable message retention means that consumers do not always need to work in real time. Messages are written to disk and will be stored with configurable retention rules. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data.  
  
📌 𝗦𝗰𝗮𝗹𝗮𝗯𝗹𝗲  
Kafka’s flexible scalability makes it easy to handle any amount of data. We can start with a single broker for POC and move to production with a large cluster easily.  
  
📌 𝗛𝗶𝗴𝗵 𝗣𝗲𝗿𝗳𝗼𝗿𝗺𝗮𝗻𝗰𝗲  
All these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease.  
  
📌 𝗣𝗹𝗮𝘁𝗳𝗼𝗿𝗺 𝗙𝗲𝗮𝘁𝘂𝗿𝗲𝘀  
These features are in the form of APIs and libraries:  
✔ 𝘒𝘢𝘧𝘬𝘢 𝘊𝘰𝘯𝘯𝘦𝘤𝘵:  
It assists with the task of pulling data from a source data system and pushing it into Kafka or pulling data from Kafka and pushing it into a sink data system.  
✔ 𝘒𝘢𝘧𝘬𝘢 𝘚𝘵𝘳𝘦𝘢𝘮𝘴:  
It provides a library for easily developing stream processing applications that are scalable and fault tolerant.

✅ Installation & Configuration of Kafka on local machine.  
  
▶ There are a few simple steps for getting Kafka up and running (without touching any configuration) 𝗼𝗻 𝗹𝗼𝗰𝗮𝗹 𝗺𝗮𝗰𝗵𝗶𝗻𝗲.  
  
📌 𝘿𝙤𝙬𝙣𝙡𝙤𝙖𝙙 𝙖𝙣𝙙 𝙄𝙣𝙨𝙩𝙖𝙡𝙡 𝙅𝘿𝙆 8.  
<https://lnkd.in/g26x-z6f>  
  
📌 𝘿𝙤𝙬𝙣𝙡𝙤𝙖𝙙 𝙩𝙝𝙚 𝙆𝙖𝙛𝙠𝙖 𝙖𝙧𝙘𝙝𝙞𝙫𝙚 𝙖𝙣𝙙 𝙚𝙭𝙩𝙧𝙖𝙘𝙩 𝙞𝙩.  
<https://lnkd.in/gX43rkjM>  
  
📌 𝙎𝙩𝙖𝙧𝙩 𝙩𝙝𝙚 𝙆𝙖𝙛𝙠𝙖 𝙚𝙣𝙫𝙞𝙧𝙤𝙣𝙢𝙚𝙣𝙩.  
✔ 𝘈𝘧𝘵𝘦𝘳 𝘦𝘹𝘵𝘳𝘢𝘤𝘵𝘪𝘯𝘨 𝘵𝘩𝘦 𝘢𝘳𝘤𝘩𝘪𝘷𝘦, 𝘳𝘶𝘯 𝘵𝘩𝘦 𝘧𝘰𝘭𝘭𝘰𝘸𝘪𝘯𝘨 𝘤𝘰𝘮𝘮𝘢𝘯𝘥 𝘪𝘯 𝘵𝘩𝘦 𝘵𝘦𝘳𝘮𝘪𝘯𝘢𝘭:   
$ bin/zookeeper-server-start[dot]sh config/zookeeper[dot]properties  
  
✔ 𝘖𝘱𝘦𝘯 𝘢𝘯𝘰𝘵𝘩𝘦𝘳 𝘵𝘦𝘳𝘮𝘪𝘯𝘢𝘭 𝘢𝘯𝘥 𝘳𝘶𝘯:  
$ bin/kafka-server-start[dot]sh config/server[dot]properties  
  
📌 Quickstart your learning by following the official docs. <https://lnkd.in/gDX8xheK>  
  
📌 The below YouTube video seems quite helpful for the beginners:  
<https://lnkd.in/gFK5vYXC>  
  
ℂ𝕣𝕖𝕕𝕚𝕥𝕤: '𝔻𝕒𝕚𝕝𝕪 ℂ𝕠𝕕𝕖 𝔹𝕦𝕗𝕗𝕖𝕣' 𝕔𝕙𝕒𝕟𝕟𝕖𝕝  
  
You should be able to set up the environment easily and play around it. You can connect to this Kakfa server from your application code and can do a lot of other stuff.  
  
So please make sure your environment is ready to use.  
   
PS: There are other ways of learning and playing with Kafka environment which are by using the Kafka docker images or on cloud. But for now, I’m skipping those as they will require some understanding of Docker and cloud technologies and will need a separate post altogether.

👉 I hope by now, you’ll have setup Kakfa in your local machine, should have created a sample topic and tested the "console producer" and "console consumer".  
  
👉 If not, please do that as next posts will contain references to the actual code which will connect to the local installation of Kafka. If there is any doubt, please leave a comment or DM.  
  
⏩ 𝗞̲𝗮̲𝗳̲𝗸̲𝗮̲ ̲𝗦̲𝗲̲𝗿̲𝗶̲𝗲̲𝘀̲ ̲(̲𝗣̲𝗮̲𝗿̲𝘁̲ ̲𝟲̲)̲  
In this post, we’ll understand more about 𝗞𝗮𝗳𝗸𝗮 𝗣𝗿𝗼𝗱𝘂𝗰𝗲𝗿.  
  
👉 Please don’t worry if you don’t understand a few terms like ‘𝘍𝘶𝘵𝘶𝘳𝘦’, ‘𝘈𝘴𝘺𝘯𝘤𝘩𝘳𝘰𝘯𝘰𝘶𝘴’, ‘𝘚𝘺𝘯𝘤𝘩𝘳𝘰𝘯𝘰𝘶𝘴’, ‘𝘚𝘦𝘳𝘪𝘢𝘭𝘪𝘻𝘦𝘳’, etc, which are used here. I’ll put brief details about these in the comments.  
  
👉 Applications need to write messages to Kafka for a variety of use cases, e.g.,:  
✔ Recording user activities from a website for auditing or analysis.  
✔ Recording metrics.  
✔ Storing log messages.  
✔ Recording information from smart devices.  
✔ Buffering information before writing to a database.  
  
𝗖̲𝗼̲𝗻̲𝘀̲𝘁̲𝗿̲𝘂̲𝗰̲𝘁̲𝗶̲𝗻̲𝗴̲ ̲𝗮̲ ̲𝗞̲𝗮̲𝗳̲𝗸̲𝗮̲ ̲𝗣̲𝗿̲𝗼̲𝗱̲𝘂̲𝗰̲𝗲̲𝗿̲  
👉 Create a Producer object with the required properties. The mandatory properties are:  
✔ 𝙗𝙤𝙤𝙩𝙨𝙩𝙧𝙖𝙥.𝙨𝙚𝙧𝙫𝙚𝙧𝙨: List of host:port pairs of brokers that the producer will use to establish initial connection to the Kafka cluster.  
✔ 𝙠𝙚𝙮.𝙨𝙚𝙧𝙞𝙖𝙡𝙞𝙯𝙚𝙧: Name of a class that will be used to serialize the keys of the records being produced to Kafka (e.g., org.apache.kafka.common.serialization.StringSerializer).  
✔ 𝙫𝙖𝙡𝙪𝙚.𝙨𝙚𝙧𝙞𝙖𝙡𝙞𝙯𝙚𝙧: Name of a class that will be used to serialize the values of the records being produced to Kafka.  
  
👉 Once we instantiate a Producer object, the below three primary methods can be used for sending messages:  
  
📌 𝗙𝗶𝗿𝗲-𝗮𝗻𝗱-𝗙𝗼𝗿𝗴𝗲𝘁:  
✔ We send a message to the server and don’t really care if it arrives successfully or not.  
✔ In case of nonretriable errors or timeout, messages will get lost, and the application will not get any information or exceptions about this.  
  
📌 𝗦𝘆𝗻𝗰𝗵𝗿𝗼𝗻𝗼𝘂𝘀:  
✔ When we send a message, the send() method returns a Future object. We use get() to wait on the Future and see if the send() was successful or not before sending the next record.  
✔ It can lead to poor performance as the thread will spend time waiting and doing nothing else.  
✔ It is usually not used in production applications.  
  
📌 𝗔𝘀𝘆𝗻𝗰𝗵𝗿𝗼𝗻𝗼𝘂𝘀:  
✔ To send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record.  
✔ We call the send() method with a callback function, which gets triggered when it receives a response from the Kafka broker.

✅ 𝑷𝒓𝒐𝒅𝒖𝒄𝒆𝒓 𝑪𝒐𝒏𝒇𝒊𝒈𝒖𝒓𝒂𝒕𝒊𝒐𝒏  
The producer has a large number of configuration parameters. However, some of the parameters have a significant impact on 𝐦𝐞𝐦𝐨𝐫𝐲 𝐮𝐬𝐞, 𝐩𝐞𝐫𝐟𝐨𝐫𝐦𝐚𝐧𝐜𝐞, 𝐚𝐧𝐝 𝐫𝐞𝐥𝐢𝐚𝐛𝐢𝐥𝐢𝐭𝐲 of the producers.  
  
⏩ Below are some of the important ones.  
📍 𝒄𝒍𝒊𝒆𝒏𝒕.𝒊𝒅:  
✔ It's a logical identifier for the client and the application it's used in.  
✔ It can be any string and will be used by the brokers to identify messages sent from the client.  
✔ It is used in logging and metrics.  
✔ Choosing a good client name will make troubleshooting much easier.  
  
📍 𝒂𝒄𝒌𝒔:  
✔ It controls how many partition replicas must receive the record before the producer can consider the write successful.  
✔ It has a significant impact on the durability of the written messages.  
  
The 'acks' parameter has the below three values.  
📌 𝒂𝒄𝒌𝒔=0:  
▶ The producer will not wait for a reply from the broker before assuming the message was sent successfully.  
▶ If something goes wrong and the broker does not receive the message, the producer will not know about it, and the message will be lost.  
▶ We can send messages as fast as the network will support (very high throughput).  
  
📌 𝒂𝒄𝒌𝒔=1:  
▶ The producer will receive a success response from the broker the moment the leader replica receives the message.  
▶ If the message can't be written to the leader, the producer will receive an error response and can retry sending the message, avoiding potential loss of data.  
▶ The message can still get lost if the leader crashes and the latest messages were not yet replicated to the new leader.  
  
📌 𝒂𝒄𝒌𝒔=𝒂𝒍𝒍:  
▶ The producer will receive a success response from the broker once all in sync replicas receive the message.  
▶ This is the safest mode since we can make sure more than one broker has the message and that the message will survive even in  
case of a crash.  
▶ The latency will be higher, since we will be waiting for more than just one broker to receive the message.

𝙋𝙧𝙤𝙙𝙪𝙘𝙚𝙧 𝘾𝙤𝙣𝙛𝙞𝙜𝙪𝙧𝙖𝙩𝙞𝙤𝙣 [𝘾𝙤𝙣𝙩𝙞𝙣𝙪𝙚𝙙...]  
  
👉 Below are some of the very important configurations related to Kafka Producers:  
  
📌 𝗯𝘂𝗳𝗳𝗲𝗿.𝗺𝗲𝗺𝗼𝗿𝘆  
It sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers.  
  
📌 𝗰𝗼𝗺𝗽𝗿𝗲𝘀𝘀𝗶𝗼𝗻.𝘁𝘆𝗽𝗲  
By default, messages are sent uncompressed. This parameter can be set to snappy, gzip, lz4, or zstd, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers.  
  
📌 𝗯𝗮𝘁𝗰𝗵.𝘀𝗶𝘇𝗲  
When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes that will be used for each batch.  
  
📌 𝗺𝗮𝘅.𝗿𝗲𝗾𝘂𝗲𝘀𝘁.𝘀𝗶𝘇𝗲  
It controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request.  
  
📌 𝗿𝗲𝗰𝗲𝗶𝘃𝗲.𝗯𝘂𝗳𝗳𝗲𝗿.𝗯𝘆𝘁𝗲𝘀 & 𝘀𝗲𝗻𝗱.𝗯𝘂𝗳𝗳𝗲𝗿.𝗯𝘆𝘁𝗲𝘀  
These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to –1, the OS defaults will be used.  
  
📌 𝗲𝗻𝗮𝗯𝗹𝗲.𝗶𝗱𝗲𝗺𝗽𝗼𝘁𝗲𝗻𝗰𝗲  
✔ A service is called idempotent if performing the same operation multiple times has the same result as performing it a single time.  
✔ Starting in version 0.11, Kafka supports exactly once semantics. Exactly-once messaging semantics with Kafka means the combined outcome of multiple steps will happen exactly-once.  
✔ A message will be consumed, processed, and resulting messages produced, exactly-once.  
  
📌 𝗺𝗮𝘅.𝗯𝗹𝗼𝗰𝗸.𝗺𝘀  
It controls how long the producer may block when calling send() and when explicitly requesting metadata via partitionsFor().  
  
📌 𝗱𝗲𝗹𝗶𝘃𝗲𝗿𝘆.𝘁𝗶𝗺𝗲𝗼𝘂𝘁.𝗺𝘀  
It will limit the amount of time spent from the point a record is ready for sending (send() returned successfully and the record is placed in a batch) until either the broker responds or the client gives up, including time spent on retries.  
  
📌 𝗿𝗲𝗾𝘂𝗲𝘀𝘁.𝘁𝗶𝗺𝗲𝗼𝘂𝘁.𝗺𝘀  
It controls how long the producer will wait for a reply from the server when sending data.  
  
📌 𝗹𝗶𝗻𝗴𝗲𝗿.𝗺𝘀  
It controls the amount of time to wait for additional messages before sending the current batch. KafkaProducer sends a batch of messages either when the current batch is full or when the linger limit is reached.  
  
PS: Next Post: 𝗦𝗲𝗿𝗶𝗮𝗹𝗶𝘇𝗲𝗿𝘀.

⏩ 𝗞̲𝗮̲𝗳̲𝗸̲𝗮̲ ̲𝗦̲𝗲̲𝗿̲𝗶̲𝗲̲𝘀̲ ̲(̲𝗣̲𝗮̲𝗿̲𝘁̲ ̲𝟵̲)̲ [GitHub link in comments]  
  
📌 𝗦𝗲𝗿𝗶𝗮𝗹𝗶𝘇𝗮𝘁𝗶𝗼𝗻.  
  
👉 Serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation.  
  
👉 It is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later.  
  
👉 In Part 6, we saw how to use String serializer in our code. Kafka also includes serializers for  
Integers, ByteArrays, and many more.  
  
👉 Let’s talk about 𝘼𝙥𝙖𝙘𝙝𝙚 𝘼𝙫𝙧𝙤, which is a data serialization system:  
✔ It provides rich data structures and simple integration with dynamic languages.  
✔ It uses JSON for defining data types/protocols and serializes data in a compact binary format.  
✔ It provides a container file, to store persistent data.  
✔ An Avro container file consists of a file header, followed by one or more file data blocks.  
  
👉 A file header consists of:  
✔ Four bytes, ASCII 'O', 'b', 'j', followed by the Avro version number which is 1 (0x01) (Binary values 0x4F 0x62 0x6A 0x01).  
✔ File metadata, including the schema definition.  
✔ The 16-byte, randomly-generated sync marker for this file.  
  
👉 We will see the code on how to write Avro records using Kafka producer and how to read those records using consumer. Before that, you need to understand 𝗦𝗰𝗵𝗲𝗺𝗮 𝗥𝗲𝗴𝗶𝘀𝘁𝗿𝘆.  
  
👉 What is Schema Registry?  
✔ It provides a centralized repository for managing and validating schemas for topic message data, and for serialization and deserilazation of the data over the network.  
✔ It’s not part of Apache Kafka, but there are several open source options to choose from.  
  
👉 We will install Schema registry in our local machine using Docker. Before that, please use the link in the comments to install Docker software on your system as we will be using Docker in the later part of the series to install any kind of software.  
  
👉 Make sure you have local instance of Kafka running. Follow 𝗣𝗮𝗿𝘁#𝟱 of the series to install and configure Kakfa on your local machine.  
  
The docker command to run Schema Registry on your local machine has been provided in comments.

 𝑹𝒆𝒂𝒅 𝒂𝒏𝒅 𝒘𝒓𝒊𝒕𝒆 𝑨𝒗𝒓𝒐 𝒅𝒂𝒕𝒂 𝒖𝒔𝒊𝒏𝒈 𝑲𝒂𝒇𝒌𝒂 𝒑𝒓𝒐𝒅𝒖𝒄𝒆𝒓𝒔 𝒂𝒏𝒅 𝒄𝒐𝒏𝒔𝒖𝒎𝒆𝒓𝒔. 🔥  
  
👉 Before that, let’s understand a bit more about Avro. Below are the benefits of serializing the data in Avro format:  
▶ Avro relies on a schema. This means every field is properly described and documented.  
▶ Avro data format is a compact binary format, so it takes less space both on a wire and on a disk.  
▶ It has support for a variety of programming languages.  
▶ In Avro, every message contains the schema used to serialize it. That means that when you’re reading messages, you always know how to deserialize them, even if the schema has changed.  
  
👉 𝐃𝐫𝐚𝐰𝐛𝐚𝐜𝐤:  
Every Avro message contains the schema used to serialize the message. So, if there is a need to send millions of messages per day to Kafka, it’s a waste of bandwidth and storage space to send the same schema information repeatedly.  
  
👉 𝐒𝐜𝐡𝐞𝐦𝐚 𝐑𝐞𝐠𝐢𝐬𝐭𝐫𝐲 𝐭𝐨 𝐭𝐡𝐞 𝐫𝐞𝐬𝐜𝐮𝐞:  
▶ It provides a RESTful interface for storing and receiving Avro schemas.   
▶ It supports schema evolution and allows us to enforce the rules for validating a schema compatibility when the schema is modified.  
  
👉 𝐇𝐨𝐰 𝐝𝐨𝐞𝐬 𝐢𝐭 𝐰𝐨𝐫𝐤?  
▶ The schema is not sent inside the Kafka record.  
▶ The producer checks whether schema already exists in the Schema Registry.  
▶ If the schema is not present, it will write the schema in the Schema Registry.  
▶ The producer will obtain the id of the schema and will send that id inside the record, saving a lot of space.  
▶ The consumer will read the message and then contact the Schema Registry with the schema id from the record to get the full schema and cache it locally.  
  
👉 Now that you are aware about Avro and Schema Registry, it’s time to see them in action.  
  
👉 Please make sure you follow the Kafka series, and your local Kafka instance is up and running (see Part-5 for details) as well as the dockerized Schema Registry is up and running (see Part-9).  
  
👉 I’m putting the 𝐆𝐢𝐭𝐇𝐮𝐛 𝐩𝐫𝐨𝐣𝐞𝐜𝐭 𝐥𝐢𝐧𝐤 𝐜𝐨𝐧𝐭𝐚𝐢𝐧𝐢𝐧𝐠 𝐭𝐡𝐞 𝐜𝐨𝐝𝐞 𝐬𝐚𝐦𝐩𝐥𝐞𝐬 𝐢𝐧 𝐜𝐨𝐦𝐦𝐞𝐧𝐭𝐬. Please don’t hesitate to reach out if you have any query.  
  
📌 𝑰𝒎𝒑𝒐𝒓𝒕𝒂𝒏𝒕 𝑵𝒐𝒕𝒆:  
The Avro serializer can only serialize Avro objects, not POJO (Plain Old Java Object). Generating Avro classes can be done either using the avro-tools.jar or the Avro Maven plugin. I have used 𝑨𝒗𝒓𝒐 𝑴𝒂𝒗𝒆𝒏 𝒑𝒍𝒖𝒈𝒊𝒏 and you can see the dependencies in pom.xml file.

🔥 What are 𝙄𝙣𝙩𝙚𝙧𝙘𝙚𝙥𝙩𝙤𝙧𝙨 in Kafka? 🔥  
📌 Interceptors are a pluggable mechanism that allows us to intercept and process messages as they are being produced or consumed by Kafka clients.  
📌 We can modify the behavior of our Kafka client application without modifying its code.  
  
➡ There are various use cases of Interceptors, such as:  
📌 𝙇𝙤𝙜𝙜𝙞𝙣𝙜: Interceptors can be used to log messages that are produced or consumed by Kafka clients. This can be useful for debugging purposes.  
📌 𝙈𝙚𝙩𝙧𝙞𝙘𝙨 𝙘𝙤𝙡𝙡𝙚𝙘𝙩𝙞𝙤𝙣: Interceptors can be used to collect metrics on the messages that are produced or consumed by Kafka clients. This can be useful for monitoring and performance tuning.  
📌 𝙎𝙚𝙘𝙪𝙧𝙞𝙩𝙮: Interceptors can be used to add or remove security-related metadata to messages as they are produced or consumed. This can be useful for enforcing security policies and preventing unauthorized access to Kafka topics.  
  
👉 To use an interceptor, we can simply 𝙘𝙤𝙣𝙛𝙞𝙜𝙪𝙧𝙚 𝙩𝙝𝙚 𝙞𝙣𝙩𝙚𝙧𝙘𝙚𝙥𝙩𝙤𝙧 𝙘𝙡𝙖𝙨𝙨 𝙞𝙣 𝙩𝙝𝙚 𝙆𝙖𝙛𝙠𝙖 𝙥𝙧𝙤𝙙𝙪𝙘𝙚𝙧 𝙤𝙧 𝙘𝙤𝙣𝙨𝙪𝙢𝙚𝙧 configuration.  
  
👉 The interceptor will then be 𝙞𝙣𝙫𝙤𝙠𝙚𝙙 𝙛𝙤𝙧 𝙚𝙫𝙚𝙧𝙮 𝙢𝙚𝙨𝙨𝙖𝙜𝙚 𝙩𝙝𝙖𝙩 𝙞𝙨 𝙨𝙚𝙣𝙩 𝙤𝙧 𝙧𝙚𝙘𝙚𝙞𝙫𝙚𝙙 by the client.  
  
👉 Interceptors should be used with caution, as 𝙩𝙝𝙚𝙮 𝙘𝙖𝙣 𝙞𝙢𝙥𝙖𝙘𝙩 𝙩𝙝𝙚 𝙥𝙚𝙧𝙛𝙤𝙧𝙢𝙖𝙣𝙘𝙚 𝙤𝙛 𝙆𝙖𝙛𝙠𝙖 𝙘𝙡𝙞𝙚𝙣𝙩𝙨.  
  
💡 Additionally, interceptors should only be used for tasks that cannot be accomplished through other means, such as by modifying the producer or consumer code directly. 💯  
  
👉 I have put the example code for Inteceptors in the GitHub. The package name is "com.ashu.tutorial.interceptors". 💻  
  
➡ All you need to do is:  
▶ Make sure your Kafka local instance is up and running.  
▶ Build the project to create JAR file (mvn clean install).  
▶ Add the JAR file to the CLASSPATH.  
▶ Create a config file (let's say producer.config) with the below content:  
interceptor.classes=com.ashu.tutorial.interceptors.CountingProducerInterceptor  
counting(dot)interceptor(dot)window(dot)size(dot)ms=10000  
▶ Create a Kafka topic:  
$kafka\_home/bin/kafka-topics[dot]sh --create --bootstrap-server localhost[colon] --topic interceptor-test  
▶ For testing, run the Kafka console producer:  
$kafka\_home/bin/kafka-console-producer[dot]sh --broker-list localhost[colon]9092 –topic interceptor-test --producer.config producer.config  
  
➡ You will immediately see the Interceptors in action.Try to produce some messages on the console, it will print like this (we have used System.out.println in our Interceptor code):  
𝗧𝗼𝘁𝗮𝗹 𝘀𝗲𝗻𝘁: 𝟰  
𝗧𝗼𝘁𝗮𝗹 𝗮𝗰𝗸𝗻𝗼𝘄𝗹𝗲𝗱𝗴𝗲𝗱: 𝟬

🔥 𝙃𝙖𝙫𝙚 𝙮𝙤𝙪 𝙝𝙚𝙖𝙧𝙙 𝙖𝙗𝙤𝙪𝙩 𝘾𝙤𝙣𝙙𝙪𝙠𝙩𝙤𝙧? 🔥  
   
🎆 Conduktor is a software platform designed to make working with Apache Kafka easier for developers, DevOps engineers, and data scientists. Some of the common use cases of Conduktor are:  
   
📌 𝗠𝗼𝗻𝗶𝘁𝗼𝗿𝗶𝗻𝗴 𝗞𝗮𝗳𝗸𝗮 𝗰𝗹𝘂𝘀𝘁𝗲𝗿𝘀:  
Conduktor provides real-time monitoring of Kafka clusters and helps users keep track of the performance and health of their Kafka clusters. Users can view important metrics such as broker and topic-level throughput, lag, and CPU usage.  
   
📌 𝗗𝗲𝗯𝘂𝗴𝗴𝗶𝗻𝗴 𝗞𝗮𝗳𝗸𝗮 𝗮𝗽𝗽𝗹𝗶𝗰𝗮𝘁𝗶𝗼𝗻𝘀:  
Conduktor provides a comprehensive set of debugging tools to help users debug Kafka applications. For example, users can view the details of individual messages, view the message history of a topic, and analyze consumer lag.  
   
📌 𝗠𝗮𝗻𝗮𝗴𝗶𝗻𝗴 𝗞𝗮𝗳𝗸𝗮 𝘁𝗼𝗽𝗶𝗰𝘀 𝗮𝗻𝗱 𝗱𝗮𝘁𝗮:  
Conduktor provides an intuitive and user-friendly interface to manage Kafka topics and data. Users can create, delete, and modify topics, as well as view and edit messages in real-time.  
   
📌 𝗦𝗲𝗰𝘂𝗿𝗶𝗻𝗴 𝗞𝗮𝗳𝗸𝗮 𝗰𝗹𝘂𝘀𝘁𝗲𝗿𝘀:   
Conduktor provides features to help users secure their Kafka clusters, such as SSL/TLS encryption and authentication, SASL authentication, and ACL management.  
   
📌 𝗗𝗲𝘃𝗲𝗹𝗼𝗽𝗶𝗻𝗴 𝗞𝗮𝗳𝗸𝗮 𝗮𝗽𝗽𝗹𝗶𝗰𝗮𝘁𝗶𝗼𝗻𝘀:  
Conduktor provides features to help users develop Kafka applications, such as the ability to create and test Kafka producers and consumers within the platform. Additionally, Conduktor provides support for various programming languages and frameworks, including Java, Python, and Spring.  
   
👉 If you have Docker software installed, you can get Conduktor up and running in no time, for testing and learning purposes.  
   
👉 I have put the steps in PDF attached in the post. It’s enough for getting started. For advanced setting and setup options, please visit the link in the comments.  
   
👉 We should emphasize a lot on the hands-on as then only we'll feel confident and it will give us the motivation to continue the learning. Otherwise, theoretical knowledge won’t help us much in the long run.