# General Questions:

## What is Kafka?

Kafka is an open source project, distributed publish-subscribe messaging system.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| ***High Throughput*** | Support for millions of messages with modest hardware |
| ***Scalability*** | Highly scalable distributed systems with no downtime |
| ***Replication*** | Messages are replicated across the cluster to provide support for multiple subscribers and balances the consumers in case of failures |
| ***Durability*** | Provides support for persistence of message to disk |
| ***Stream Processing*** | Used with real-time streaming applications like Apache Spark & Storm |
| ***Data Loss*** | Kafka with proper configurations can ensure zero data loss |

## List the various component in Kafa?

* Topic – a stream of messages belonging to the same type
* Producer – that can publish messages to a topic
* Brokers – a set of servers where the publishes messages are stored
* Consumer – that subscribes to various topics and pulls data from the brokers.

## Kafka vs traditional Messaging?

* **Fast:**A single Kafka broker can serve thousands of clients by handling megabytes of reads and writes per second
* **Scalable:** Data are partitioned and streamlined over a cluster of machines to enable larger data
* **Durable**: Messages are persistent and is replicated within the cluster to prevent data loss
* **Distributed by Design:** It provides fault tolerance guarantees and durability

### 1. Design

**Apache Kafka :** **client-centric**, with the client taking over many of the functions of a traditional broker, such as fair distribution of related messages to consumers, in return for an extremely fast and scalable broker.

**Traditional message brokers :** **broker-centric**, where the broker is responsible for the distribution of messages, and clients only have to worry about sending and receiving messages.

### 2. Scalability

**Apache Kafka :** **horizontally scalable** by adding more partitions  
**Traditional message brokers :** not possible to scale horizontally

### 3. Performance

**Apache Kafka :** does not slow down with addition of new consumers  
**Traditional message brokers :** both queue and topic performance degrades as the number of consumers rise on a destination

### 4. Publish options

**Apache Kafka :** unifies both publish-subscribe and point-to-point messaging under a single destination type— **topic**  
  
**Traditonal message brokers :** provides both **publish-subscribe** and **point-to-point messaging** for publishing messages

### 5. Journal

**Apache Kafka :** each topic has a journal and there can be multiple topics.  
**Traditional message brokers :** messages from all queues are stored in the same journal

### 6. Message delivery failures

**Apache Kafka :** client is responsible for replaying failed messages  
**Traditional message brokers :** broker is responsible for re-delivery of messages

### 7. Message structure

**Apache Kafka :** message is a key-value pair. Payload of the message is sent as the value. Key, on the other hand, is used primarily for partitioning purposes and should contain a business-specific key in order to place related messages on the same partition.  
  
**Traditional message brokers :** message consists of metadata (headers and properties) and body (payload)

### 8. Message integrity

**Apache Kafka :** includes checksums to detect corruption of messages in storage and has a comprehensive set of security features  
  
**Traditonal message brokers :** no such option provided out-of-the box

### 9. Message retention

**Apache Kafka :** messages are **not** deleted on the broker once consumed.  
  
**Traditional message brokers :** broker would either delete a successfully processed message or re-deliver an unprocessed or failed one (assuming transactions were in play)

### 10. Replay of messages

**Apache Kafka :** since Kafka retains messages, it is possible for clients to retrieve any message, providing option for re-processing of all messages.  
  
**Traditional message brokers :** not possible, since, broker does not re-deliver messages that have been processed successfully by client

## What is topic?

### A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the Kafka cluster maintains a partitioned log.

## What is Offset?

### Messages contained in the partitions are assigned a unique ID number that is called the offset. The role of the offset is to uniquely identify every message within the partition.

## What is a consumer group?

### Consumer Groups is a concept exclusive to Kafka.  Every Kafka consumer group consists of one or more consumers that jointly consume a set of subscribed topics.

## Can we use kafka without zookeeper?

### No, it is not possible to bypass Zookeeper and connect directly to the Kafka server. If, for some reason, ZooKeeper is down, you cannot service any client request.

## Explain Leader and follower?

### Every partition in Kafka has one server which plays the role of a Leader, and none or more servers that act as Followers. The Leader performs the task of all read and write requests for the partition, while the role of the Followers is to passively replicate the leader. In the event of the Leader failing, one of the Followers will take on the role of the Leader. This ensures load balancing of the server.

## Replicas and ISR?

### Replicas are essentially a list of nodes that replicate the log for a particular partition irrespective of whether they play the role of the Leader. On the other hand, ISR stands for In-Sync Replicas. It is essentially a set of message replicas that are synced to the leaders.

## Why replicas are important?

### Replication ensures that published messages are not lost and can be consumed in the event of any machine error, program error or frequent software upgrades.

## If a replica out of ISR, what does that mean?

### It means that the Follower is unable to fetch data as fast as data accumulated by the Leader.

## How to start a kafka server?

### Since Kafka uses ZooKeeper, it is essential to initialize the ZooKeeper server, and then fire up the Kafka server.

### To start the ZooKeeper server: > bin/zookeeper-server-start.sh config/zookeeper.properties

### Next, to start the Kafka server: > bin/kafka-server-start.sh config/server.properties

## Partioning key?

### Within the Producer, the role of a Partitioning Key is to indicate the destination partition of the message. By default, a hashing-based Partitioner is used to determine the partition ID given the key. Alternatively, users can also use customized Partitions.

## QueueFullException error?

### QueueFullException typically occurs when the Producer attempts to send messages at a pace that the Broker cannot handle. Since the Producer doesn’t block, users will need to add enough brokers to collaboratively handle the increased load.

## Kafka vs Flume?

### Even though both are used for real-time processing, Kafka is scalable and ensures message durability.

### Flume’s major use-case is to gulp down the data into Hadoop. The Flume is incorporated with the Hadoop’s monitoring system, file formats, file system and utilities such as Morphlines. Flume’s design of sinks, sources and channels mean that with the aid of Flume one can shift data among other systems lithely, but the main feature is its Hadoop integration.

### The Flume is the best option used when you have non-relational data sources if you have a long file to stream into the Hadoop.

### Kafka’s major use-case is a distributed publish-subscribe messaging system. Kafka is not developed specifically for Hadoop and using Kafka to read and write data to Hadoop is considerably trickier than it is in Flume.

### Kafka can be used when you particularly need a highly reliable and scalable enterprise messaging system to connect many multiple systems like Hadoop.

## Main api of Kafka?

Producer api

Consumer api

Connector api

Kafka stream api

## How kafka ensure load balancing?

### As the main role of the Leader is to perform the task of all read and write requests for the partition, whereas Followers passively replicate the leader. Hence, at the time of Leader failing, one of the Followers takeover the role of the Leader. Basically, this entire process ensures load balancing of the servers.

## Retention period

However, retention period retains all the published records within the Kafka cluster. It doesn’t check whether they have been consumed or not. Moreover, the records can be discarded by using a configuration setting for the retention period. And, it results as it can free up some space.

By default value is 168 hours.

Log.retention.ms always take precedence over log.retention.hours or log.retention.minutes.

## Maximum size that can be received by kafka?

The maximum size of a message that can be received by the Kafka is approx. 1000000 bytes. That is 1 MB. Message.max.bytes

## How to send large massege over kafka?

* Consumer side:fetch.message.max.bytes - this will determine the largest size of a message that can be fetched by the consumer.
* Broker side: replica.fetch.max.bytes - this will allow for the replicas in the brokers to send messages within the cluster and make sure the messages are replicated correctly. If this is too small, then the message will never be replicated, and therefore, the consumer will never see the message because the message will never be committed (fully replicated).
* Broker side: message.max.bytes - this is the largest size of the message that can be received by the broker from a producer.
* Broker side (per topic): max.message.bytes - this is the largest size of the message the broker will allow to be appended to the topic. This size is validated pre-compression. (Defaults to broker's message.max.bytes.)

I found out the hard way about number 2 - you don't get ANY exceptions, messages, or warnings from Kafka, so be sure to consider this when you are sending large messages.

For kafka 0.10 and above

### Broker: No changes, you still need to increase properties message.max.bytes and replica.fetch.max.bytes. message.max.bytes has to be equal or smaller(\*) than replica.fetch.max.bytes.

### Producer: Increase max.request.size to send the larger message.

### Consumer: Increase max.partition.fetch.bytes to receive larger messages.

### (\*) Read the comments to learn more about message.max.bytes<=replica.fetch.max.bytes

## Geo replication?

### For our cluster, Kafka MirrorMaker offers geo-replication. Basically, messages are replicated across multiple data centers or cloud regions, with MirrorMaker. So, it can be used in active/passive scenarios for backup and recovery; or also to place data closer to our users, or support data locality requirements.

## Multi tenancy?

## Producer API

he role of Kafka’s Producer API is to wrap the two producers – kafka.producer.SyncProducer and the kafka.producer.async.AsyncProducer.

The goal is to expose all the producer functionality through a single API to the client.

## Consumer API

Kafka provides single consumer abstractions that discover both queuing and publish-subscribe Consumer Group. They tag themselves with a user group and every communication available on a topic is distributed to one user case within every promising user group. User instances are in disconnected process. We can determine the messaging model of the consumer based on the consumer groups.

* If all consumer instances have the same consumer set, then this works like a conventional queue adjusting load over the consumers.
* If all customer instances have dissimilar consumer groups, then this works like a publish-subscribe and all messages are transmitted to all the consumers.

## Connect API

## Sream API

## Kafka vs Rabbit MQ

Apache Kafka and RabbitMQ are two open-source and commercially-supported pub/sub systems, readily adopted by enterprises. RabbitMQ is an older tool released in 2007 and was a primary component in messaging and SOA systems. Today it is also being used for streaming use cases. Kafka is a newer tool, released in 2011, which, from the onset, was built for streaming scenarios.

RabbitMQ is a general purpose message broker that supports protocols including, MQTT, AMQP, and STOMP. It can deal with high-throughput use cases, such as online payment processing. It can handle background jobs or act as a message broker between microservices.

Kafka is a message bus developed for high-ingress data replay and streams. Kafka is a durable message broker that enables applications to process, persist and re-process streamed data. Kafka has a straightforward routing approach that uses a routing key to send messages to a topic.

### Kafka vs RabbitMQ - Differences in Architecture

#### **RabbitMQ Architecture**

* **General purpose message broker**—uses variations of request/reply, point to point, and pub-sub communication patterns.
* **Smart broker / dumb consumer model**—consistent delivery of messages to consumers, at around the same speed as the broker monitors the consumer state.
* **Mature platform**—well supported, available for Java, client libraries, .NET, Ruby, node.js. Offers dozens of plugins.
* **Communication**—can be synchronous or asynchronous.
* **Deployment scenarios**—provides distributed deployment scenarios.
* **Multi-node cluster to cluster federation**—does not rely on external services, however, specific cluster formation plugins can use DNS, APIs, Consul, etc.

#### **Apache Kafka Architecture**

* **High volume publish-subscribe messages and streams platform**—durable, fast and scalable.
* **Durable message store**—like a log, run in a server cluster, which keeps streams of records in topics (categories).
* **Messages**—made up of a value, a key and a timestamp.
* **Dumb broker / smart consumer model**—does not try to track which messages are read by consumers and only keeps unread messages. Kafka keeps all messages for a set period of time.
* **Requires external services to run**—in some cases Apache Zookeeper.

[](https://www.upsolver.com/blog/aws-athena-performance-best-practices-performance-tuning-tips)

### Pull vs Push Approach

#### **Apache Kafka: Pull-based approach**

Kafka uses a pull model. Consumers request batches of messages from a specific offset. Kafka permits long-pooling, which prevents tight loops when there is no message past the offset.

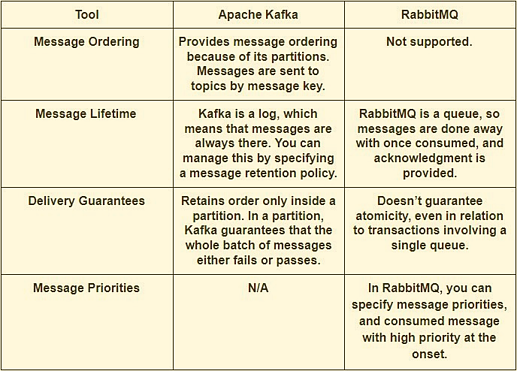
A pull model is logical for Kafka because of its partitions. Kafka provides message order in a partition with no contending consumers. This allows users to leverage the batching of messages for effective message delivery and higher throughput.

#### **RabbitMQ: Push-based approach**

RabbitMQ uses a push model and stops overwhelming consumers through a prefetch limit defined on the consumer. This can be used for low latency messaging.

The aim of the push model is to distribute messages individually and quickly, to ensure that work is parallelized evenly and that messages are processed approximately in the order in which they arrived in the queue.

### How Do They Handle Messaging?



### Kafka vs RabbitMQ Performance

#### **Apache Kafka:**

Kafka offers much higher performance than message brokers like RabbitMQ. It uses sequential disk I/O to boost performance, making it a suitable option for implementing queues. It can achieve high throughput (millions of messages per second) with limited resources, a necessity for big data use cases.

#### **RabbitMQ:**

RabbitMQ can also process a million messages per second but requires more resources (around 30 nodes). You can use RabbitMQ for many of the same use cases as Kafka, but you’ll need to combine it with other tools like Apache Cassandra.

### What are the Best Use Cases?

#### **Apache Kafka Use Cases**

Apache Kafka provides the broker itself and has been designed towards stream processing scenarios. Recently, it has added Kafka Streams, a client library for building applications and microservices. Apache Kafka supports use cases such as metrics, activity tracking, log aggregation, stream processing, commit logs and event sourcing.

The following messaging scenarios are especially suited for Kafka:

* Streams with complex routing, throughput of 100K/sec events or more, with “at least once” partitioned ordering
* Applications requiring a stream history, delivered in “at least once” partitioned ordering. Clients can see a ‘replay’ of the event stream.
* Event sourcing, modeling changes to a system as a sequence of events.
* Stream processing data in multi-stage pipelines. The pipelines generate graphs of real-time data flows.

#### **RabbitMQ Use Cases**

RabbitMQ can be used when web servers need to quickly respond to requests. This eliminates the need to perform resource-intensive activities while the user waits for a result. RabbitMQ is also used to convey a message to various recipients for consumption or to share loads between workers under high load (20K+ messages/second).

Scenarios that RabbitMQ can be used for:

* Applications that need to support legacy protocols, such as STOMP, MQTT, AMQP, 0-9-1.
* Granular control over consistency/set of guarantees on a per-message basis
* Complex routing to consumers
* Applications that need a variety of publish/subscribe, point-to-point request/reply messaging capabilities.

Kafka and RabbitMQ: Summing Up

This guide has covered the major differences and similarities between Apache Kafka and RabbitMQ. Both can consume several millions of messages per second, though their architectures differ, and each performs better in certain environments. RabbitMQ controls its messages almost in-memory, using big cluster (30+ nodes). Comparatively, Kafka leverages sequential disk I/O operations and thus demands less hardware.

## Kafka cluster?

In order to overcome the challenges of collecting the large volume of data, and analyzing the collected data we need a messaging system. Hence Apache Kafka came in the story. Its benefits are:

* It is possible to track web activities just by storing/sending the events for real-time processes.
* Through this, we can Alert as well as report the operational metrics.
* Also, we can transform data into the standard format.
* Moreover, it allows continuous processing of streaming data to the topics.

Due to its this wide use, it is ruling over some of the most popular applications like ActiveMQ, RabbitMQ, AWS etc.

## Log and data log in kafka?

## How to tune kafka ?

<https://data-flair.training/blogs/kafka-performance-tuning/>

## Disadvantage of Kafka

<https://data-flair.training/blogs/advantages-and-disadvantages-of-kafka/>

## Kafka Guarantee?

### The order will be same for both the Messages sent by a producer to a particular topic partition. That

### Moreover, the consumer instance sees records in the order in which they are stored in the log.

### Also, we can tolerate up to N-1 server failures, even without losing any records committed to the log.

## Role of zookeeper?

### The basic responsibility of Zookeeper is to build coordination between different nodes in a cluster.

### Since Zookeeper works as periodically commit offset so that if any node fails, it will be used to recover from previously committed to offset.

### The ZooKeeper is also responsible for configuration management, leader detection, detecting if any node leaves or joins the cluster, synchronization, etc.

### Kafka uses Zookeeper to store offsets of messages consumed for a specific topic and partition by a specific Consumer Group.

## How does partition assignment work by broker?

### When a consumer wants to join a group, it sends a JoinGroup request to the group coordinator. The first consumer to join the group becomes the group leader. The leader receives a list of all consumers in the group from the group coordinator and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of PartitionAssignor to decide which partitions should be handled by which consumer.

### After deciding on the partition assignment, the consumer group leader sends the list of assignments to the Group Coordinator, which sends this information to all the consumers. Each consumer only sees his own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

## How to ensure exactly one message?

### During data, production to get exactly once messaging from Kafka you have to follow two things avoiding duplicates during data consumption and avoiding duplication during data production. Here are the two ways to get exactly one semantics while data production:

### Avail a single writer per partition, every time you get a network error checks the last message in that partition to see if your last write succeeded

### In the message include a primary key (UUID or something) and de-duplicate on the consumer.

## It is possible to get the message offset after producing?

You cannot do that from a class that behaves as a producer like in most queue systems, its role is to fire and forget the messages. The broker will do the rest of the work like appropriate metadata handling with id’s, offsets, etc.

As a consumer of the message, you can get the offset from a Kafka broker. If you gaze in the **SimpleConsumer** class, you will notice it fetches **MultiFetchResponse** objects that include offsets as a list. In addition to that, when you iterate the Kafka Message, you will have **MessageAndOffset** objects that include both, the offset and the message sent.

# References Link:

<https://data-flair.training/blogs/kafka-interview-questions/>

<https://kafka.apache.org/intro>