KAFKA_IQ

1. **What is Apache Kafka?**
   Apache Kafka is a publish-subscribe **open source** message broker application. This messaging application was coded in “Scala”. Basically, this project was started by the Apache software. Kafka’s design pattern is mainly based on the transactional logs design.
2. **Enlist the several components in Kafka**
   The most important elements of Kafka are:

- Topic – Kafka Topic is the bunch or a collection of messages.
- Producer – In Kafka, Producers issue communications as well as publishes messages to a Kafka topic.
- Consumer – Kafka Consumers subscribes to a topic(s) and also reads and processes messages from the topic(s).
- Brokers –While it comes to manage storage of messages in the topic(s) we use Kafka Brokers.

3. **Explain the role of the offset**
   There is a sequential ID number given to the messages in the partitions what we call, an offset. So, to identify each message in the partition uniquely, we use these offsets.
4. **What is a Consumer Group?**
   The concept of Consumer Groups is exclusive to Apache Kafka. Basically, every kafka cosumer group consists of one or more consumers that jointly consume a set of subscribed topics
5. **What is the role of the ZooKeeper in Kafka?**
   Apache Kafka is a distributed system is built to use Zookeeper. Although, Zookeeper’s main role here is to build coordination between different nodes in a cluster. However, we also use Zookeeper to recover from previously committed offset if any node fails because it works as periodically commit offset.
6. **Why is Kafka technology significant to use?**
   There are some advantages of Kafka, which makes it significant to use:

- High-throughput : We do not need any large hardware in Kafka, because it is capable of handling high-velocity and high-volume data. Moreover, it can also support message throughput of thousands of messages per second.
- Low Latency : Kafka can easily handle these messages with the very low latency of the range of milliseconds, demanded by most of the new use cases.
- Fault-Tolerant : Kafka is resistant to node/machine failure within a cluster.

- Durability : As Kafka supports messages replication, so, messages are never lost. It is one of the reasons behind durability.
- Scalability : Kafka can be scaled-out, without incurring any downtime on the fly by adding additional nodes.

7. **What are consumers or users?**
   Kafka Consumer subscribes to a topic(s), and also reads and processes messages from the topic(s). Moreover, with a consumer group name, Consumers label themselves. In other words, within each subscribing consumer group, each record published to a topic is delivered to one consumer instance. Make sure it is possible that Consumer instances can be in separate processes or on separate machines.
8. **What ensures load balancing of the server in Kafka?**
   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.
9. **What roles do Replicas and the ISR play?**
   Basically, a list of nodes that replicate the log is Replicas. Especially, for a particular partition. However, they are irrespective of whether they play the role of the Leader. In addition, ISR refers to In-Sync Replicas. On defining ISR, it is a set of message replicas that are synced to the leaders.
10. **Why are Replications critical in Kafka?**
    Because of Replication, we can be sure that published messages are not lost and can be consumed in the event of any machine error, program error or frequent software upgrades.
11. **In the Producer, when does QueueFullException occur?**
    whenever the Kafka Producer attempts to send messages at a pace that the Broker cannot handle at that time QueueFullException typically occurs. However, to collaboratively handle the increased load, users will need to add enough brokers(servers, nodes), since the Producer doesn't block.
12. **What are the types of traditional method of message transfer?**
    Basically, there are two methods of the traditional message transfer method, such as:

- Queuing: It is a method in which a pool of consumers may read a message from the server and each message goes to one of them.
- Publish-Subscribe: Whereas in Publish-Subscribe, messages are broadcasted to all consumers.

13. **What is Geo-Replication in Kafka?**
    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.

14. **Compare: RabbitMQ vs Apache Kafka**
    One of the Apache Kafka's alternative is RabbitMQ. So, let's compare both:

- Features
  - Apache Kafka– Kafka is distributed, durable and highly available, here the data is shared as well as replicated.
  - RabbitMQ– There are no such features in RabbitMQ.
- Performance rate
  - Apache Kafka– To the tune of 100,000 messages/second.
  - RabbitMQ- In case of RabbitMQ, the performance rate is around 20,000 messages/second.

15. **Compare: Traditional queuing systems vs Apache Kafka**

- Traditional queuing systems– It deletes the messages just after processing completion typically from the end of the queue.
- Apache Kafka– But in Kafka, messages persist even after being processed. That implies messages in Kafka don't get removed as consumers receive them. Logic-based processing
- Traditional queuing systems–Traditional queuing systems don't permit to process logic based on similar messages or events.
- Apache Kafka– Kafka permits to process logic based on similar messages or events.

16. **What is the benefits of Apache Kafka over the traditional technique?**

- Scalability: Kafka is designed for horizontal scalability. It can scale out by adding more brokers (servers) to the Kafka cluster to handle more partitions and thereby increase throughput. This scalability is seamless and can handle petabytes of data without downtime.
- Performance: Kafka provides high throughput for both publishing and subscribing to messages, even with very large volumes of data. It uses a disk structure that optimizes for batched writes and reads, significantly outperforming traditional databases in scenarios that involve high-volume, high-velocity data.
- Durability and Reliability: Kafka replicates data across multiple nodes, ensuring that data is not lost even if some brokers fail. This replication is configurable, allowing users to balance between redundancy and performance based on their requirements.
- Fault Tolerance: Kafka is designed to be fault-tolerant. The distributed nature of Kafka, combined with its replication mechanisms, ensures that the system continues to operate

even when individual components .

- Real-time Processing: Kafka enables real-time data processing by allowing producers to write data into Kafka topics and consumers to read data from these topics with minimal latency. This capability is critical for applications that require real-time analytics, monitoring, and response.
- Decoupling of Data Streams: Kafka allows producers and consumers to operate independently. Producers can write data to Kafka topics without being concerned about how the data will be processed. Similarly, consumers can read data from topics without needing to coordinate with producers. This decoupling simplifies system architecture and enhances flexibility.
- Replayability: Kafka stores data for a configurable period, enabling applications to replay historical data. This is valuable for new applications that need access to historical data or for recovering from errors by reprocessing data.
- High Availability: Kafka's distributed nature and replication model ensure high availability. Even if some brokers or partitions become unavailable, the system can continue to function, ensuring continuous operation of critical applications.

17. **What is the meaning of broker in Apache Kafka?**
    a broker refers to a server in the Kafka cluster that stores and manages the data. Each broker holds a set of topic partitions, allowing Kafka to efficiently handle large volumes of data by distributing the load across multiple brokers in the cluster. Brokers handle all read and write requests from Kafka producers and consumers and ensure data replication and fault tolerance to prevent data loss.
18. **What is the maximum size of a message that kafka can receive?**
    The maximum size of a message that Kafka can receive is determined by the message.max.bytes configuration parameter for the broker and the max.message.bytes parameter for the topic. By default, Kafka allows messages up to 1 MB (1,048,576 bytes) in size, but both parameters can be adjusted to allow larger messages if needed.
19. **What is the Zookeeper's role in Kafka's ecosystem and can we use Kafka without Zookeeper?**
    Zookeeper in Kafka is used for managing and coordinating Kafka brokers. It helps in leader election for partitions, cluster membership, and configuration management among other tasks. Historically, Kafka required Zookeeper to function.
    However, with the introduction of KRaft mode (Kafka Raft Metadata mode), it's possible to use Kafka without Zookeeper. KRaft mode replaces Zookeeper by using a built-in consensus mechanism for managing cluster metadata, simplifying the architecture and potentially improving performance and scalability.
20. **How are messages consumed by a consumer in apache Kafka?**
    In Apache Kafka, messages are consumed by a consumer through a pull-based model. The consumer subscribes to one or more topics and polls the Kafka broker at regular

intervals to fetch new messages. Messages are consumed in the order they are stored in the topic's partitions. Each consumer keeps track of its offset in each partition, which is the position of the next message to be consumed, allowing it to pick up where it left off across restarts or failures.

21. **How can you improve the throughput of a remote consumer?**

- Increase Bandwidth: Ensure the network connection has sufficient bandwidth to handle the data being consumed.
- Optimize Data Serialization: Use efficient data serialization formats to reduce the size of the data being transmitted.
- Concurrency: Implement concurrency in the consumer to process data in parallel, if possible.
- Batch Processing: Where applicable, batch data together to reduce the number of round-trip times needed.
- Caching: Cache frequently accessed data on the consumer side to reduce data retrieval times.
- Compression: Compress data before transmission to reduce the amount of data being sent over the network.
- Optimize Network Routes: Use optimized network paths or CDN services to reduce latency.
- Adjust Timeouts and Buffer Sizes: Fine-tune network settings, including timeouts and buffer sizes, for optimal data transfer rates.

22. **How can get Exactly-Once Messaging from Kafka during data production?**
23. Enable Idempotence: Configure the producer for idempotence by setting `enable.idempotence` to `true`. This ensures that messages are not duplicated during network errors.
24. Transactional API: Use Kafka's Transactional API by initiating transactions on the producer. This involves setting the `transactional.id` configuration and managing transactions with `beginTransaction()`, `commitTransaction()`, and `abortTransaction()` methods. It ensures that either all messages in a transaction are successfully published, or none are in case of failure, thereby achieving exactly-once semantics.
25. Proper Configuration: Alongside enabling idempotence, adjust `acks` to `all` (or `-1`) to ensure all replicas acknowledge the messages, and set an appropriate `retries` and `max.in.flight.requests.per.connection` (should be `1` when transactions are used) to handle retries without message duplication.
26. Consistent Partitioning: Ensure that messages are partitioned consistently if the order matters. This might involve custom partitioning strategies to avoid shuffling messages among partitions upon retries.

27. **What is In-Sync Messages(ISR) in Apache Kafka?**
    In Apache Kafka, ISR stands for In-Sync Replicas. It's a concept related to Kafka's high availability and fault tolerance mechanisms.
    For each partition, Kafka maintains a list of replicas that are considered "in-sync" with the leader replica. The leader replica is the one that handles all read and write requests for a specific partition, while the follower replicas replicate the leader's log. Followers that have fully caught up with the leader log are considered in-sync. This means they have replicated all messages up to the last message acknowledged by the leader.
    The ISR ensures data durability and availability. If the leader fails, Kafka can elect a new leader from the in-sync replicas, minimizing data loss and downtime.
28. **How can we reduce chrun(frequent changes) in ISR?**

- Optimize Network Configuration: Ensure that the network connections between brokers are stable and have sufficient bandwidth. Network issues can cause followers to fall behind and drop out of the ISR.
- Adjust Replica Lag Configuration: Kafka allows configuration of parameters like replica.lag.time.max.ms which defines how long a replica can be behind the leader before it is considered out of sync. Adjusting this value can help manage ISR churn by allowing replicas more or less time to catch up.
- Monitor and Scale Resources Appropriately: Ensure that all brokers have sufficient resources (CPU, memory, disk I/O) to handle their workload. Overloaded brokers may struggle to keep up, leading to replicas falling out of the ISR.
- Use Dedicated Networks for Replication Traffic: If possible, use a dedicated network for replication traffic. This can help prevent replication traffic from being impacted by other network loads.

25. **When does a broker leave ISR?**
    A broker may leave the ISR for a few reasons:

- Falling Behind: If a replica falls behind the leader by more than the configured thresholds (replica.lag.time.max.ms or replica.lag.max.messages), it is removed from the ISR.
- Broker Failure: If a broker crashes or is otherwise disconnected from the cluster, its replicas are removed from the ISR.
- Manual Intervention: An administrator can manually remove a replica from the ISR, although this is not common practice and should be done with caution.

26. **What does it indicate if replica stays out of Isr for a long time?**
    If a replica stays out of the ISR (In-Sync Replicas) for a long time, it indicates that the replica is not able to keep up with the leader's log updates. This can be due to network issues, hardware failure, or high load on the broker. As a result, the replica might become

a bottleneck for partition availability and durability, since it cannot participate in acknowledging writes or be elected as a leader if the current leader fails.

27. **What happens if the preferred replica is not in the ISR list?**
    If the preferred replica is not in the In-Sync Replicas (ISR) for a Kafka topic, the producer will either wait for the preferred replica to become available (if configured with certain ack settings) or send messages to another available broker that is part of the ISR. This ensures data integrity by only using replicas that are fully up-to-date with the leader. Consumers might experience a delay in data availability if they are set to consume only from the preferred replica and it is not available
28. **Is it possible to get the message offset after producing to a topic?**
    Yes, it is possible to get the message offset after producing a message in Kafka. When you send a message to a Kafka topic, the producer API can return metadata about the message, including the offset of the message in the topic partition
29. **What is the role of the offset in kafka?**
    In Kafka, the offset is a unique identifier for each record within a Kafka topic's partition. It denotes the position of a record within the partition. The offset is used by consumers to track which records have been read and which haven't, allowing for fault-tolerant and scalable message consumption. Essentially, it enables consumers to pick up reading from the exact point they left off, even in the event of a failure or restart, thereby ensuring that no messages are lost or read multiple times.
30. **Can you explain the concept of `leader` and `follower` in kafka ecosystem?**
    In Apache Kafka, the concepts of "leader" and "follower" refer to roles that brokers play within a Kafka cluster to manage partitions of a topic.

- Leader: For each partition of a topic, there is one broker that acts as the leader. The leader is responsible for handling all read and write requests for that partition. When messages are produced to a partition, they are sent to the leader broker, which then writes the messages to its local storage. The leader broker ensures that messages are stored in the order they are received.
- Follower: Followers are other brokers in the cluster that replicate the data of the leader for fault tolerance. Each follower continuously pulls messages from the leader to stay up-to-date, ensuring that it has an exact copy of the leader's data. In case the leader broker fails, one of the followers can be elected as the new leader, ensuring high availability.

31. **What do you mean by zookeeper in Kafka and what are its uses?**
    Apache ZooKeeper is a naming registry for distributed applications as well as a distributed, open-source configuration and synchronization service. It keeps track of the Kafka cluster nodes' status, as well as Kafka topics, partitions, and so on.
    ZooKeeper is used by Kafka brokers to maintain and coordinate the Kafka cluster. When the topology of the Kafka cluster changes, such as when brokers and topics are added or

removed, ZooKeeper notifies all nodes. When a new broker enters the cluster, for example, ZooKeeper notifies the cluster, as well as when a broker fails. ZooKeeper also allows brokers and topic partition pairs to elect leaders, allowing them to select which broker will be the leader for a given partition (and server read and write operations from producers and consumers), as well as which brokers contain clones of the same data. When the cluster of brokers receives a notification from ZooKeeper, they immediately begin to coordinate with one another and elect any new partition leaders that are required. This safeguards against the unexpected absence of a broker.

32. **What do you mean by a Partition in Kafka?**
    Kafka topics are separated into partitions, each of which contains records in a fixed order. A unique offset is assigned and attributed to each record in a partition. Multiple partition logs can be found in a single topic. This allows several users to read from the same topic at the same time. Topics can be parallelized via partitions, which split data into a single topic among numerous brokers.
    Replication in Kafka is done at the partition level. A replica is the redundant element of a topic partition. Each partition often contains one or more replicas, which means that partitions contain messages that are duplicated across many Kafka brokers in the cluster. One server serves as the leader of each partition (replica), while the others function as followers. The leader replica is in charge of all read-write requests for the partition, while the followers replicate the leader. If the lead server goes down, one of the followers takes over as the leader. To disperse the burden, we should aim for a good balance of leaders, with each broker leading an equal number of partitions.
33. **What do you mean by Kafka schema registry?**
    A Schema Registry is present for both producers and consumers in a Kafka cluster, and it holds Avro schemas. For easy serialization and de-serialization, Avro schemas enable the configuration of compatibility parameters between producers and consumers. The Kafka Schema Registry is used to ensure that the schema used by the consumer and the schema used by the producer are identical. The producers just need to submit the schema ID and not the whole schema when using the Confluent schema registry in Kafka. The consumer looks up the matching schema in the Schema Registry using the schema ID.
34. **Tell me about some of the use cases where Kafka is not suitable.**
    Following are some of the use cases where Kafka is not suitable

- Kafka is designed to manage large amounts of data. Traditional messaging systems would be more appropriate if only a small number of messages need to be processed every day.
- Although Kafka includes a streaming API, it is insufficient for executing data transformations. For ETL (extract, transform, load) jobs, Kafka should be avoided.
- There are superior options, such as RabbitMQ, for scenarios when a simple task queue is required.

- If long-term storage is necessary, Kafka is not a good choice. It simply allows you to save data for a specific retention period and no longer.

35. **What do you understand about Kafka MirrorMaker?**
    The MirrorMaker is a standalone utility for copying data from one Apache Kafka cluster to another. The MirrorMaker reads data from original cluster topics and writes it to a destination cluster with the same topic name. The source and destination clusters are separate entities that can have various partition counts and offset values.
36. **Describe message compression in Kafka. What is the need of message compression in Kafka? Also mention if there are any disadvantages of it.**
    Producers transmit data to brokers in JSON format in Kafka. The JSON format stores data in string form, which can result in several duplicate records being stored in the Kafka topic. As a result, the amount of disc space used increases. As a result, before delivering messages to Kafka, compression or delaying of data is performed to save disk space. Because message compression is performed on the producer side, no changes to the consumer or broker setup are required.
    It is advantageous because of the following factors:

- It decreases the latency of messages transmitted to Kafka by reducing their size.
- Producers can send more net messages to the broker with less bandwidth.
- When data is saved in Kafka using cloud platforms, it can save money in circumstances where cloud services are paid.
- Message compression reduces the amount of data stored on disk, allowing for faster read and write operations.
  Message Compression has the following disadvantages :
- Producers must use some CPU cycles to compress their work.
- Decompression takes up several CPU cycles for consumers.
- Compression and decompression place a higher burden on the CPU.

37. **What do you understand about log compaction and quotas in Kafka?**
    Log compaction is a way through which Kafka assures that for each topic partition, at least the last known value for each message key within the log of data is kept. This allows for the restoration of state following an application crash or a system failure. During any operational maintenance, it allows refreshing caches after an application restarts. Any consumer processing the log from the beginning will be able to see at least the final state of all records in the order in which they were written, because of the log compaction.
    A Kafka cluster can apply quotas on producers and fetch requests as of Kafka 0.9. Quotas are byte-rate limits that are set for each client-id. A client-id is a logical identifier for a request-making application. A single client-id can therefore link to numerous producers and client instances. The quota will be applied to them all as a single unit. Quotas prevent

a single application from monopolizing broker resources and causing network saturation by consuming extremely large amounts of data.

38. **What do you mean by an unbalanced cluster in Kafka? How can you balance it?**
    It's as simple as assigning a unique broker id, listeners, and log directory to the server.properties file to add new brokers to an existing Kafka cluster. However, these brokers will not be allocated any data partitions from the cluster's existing topics, so they won't be performing much work unless the partitions are moved or new topics are formed.
    A cluster is referred to as unbalanced if it has any of the following problems :
    Leader Skew:
    Broker Skew:
39. **What do you mean by BufferExhaustedException and OutOfMemoryException in Kafka?**
    When the producer can't assign memory to a record because the buffer is full, a **BufferExhaustedException** is thrown. If the producer is in non-blocking mode, and the rate of production exceeds the rate at which data is transferred from the buffer for long enough, the allocated buffer will be depleted, the exception will be thrown.
    If the consumers are sending huge messages or if there is a spike in the number of messages sent at a rate quicker than the rate of downstream processing, an **OutOfMemoryException** may arise. As a result, the message queue fills up, consuming memory space.
40. **What are Znodes in Kafka Zookeeper? How many types of Znodes are there?**
    The nodes in a ZooKeeper tree are called znodes. Version numbers for data modifications, ACL changes, and timestamps are kept by Znodes in a structure. ZooKeeper uses the version number and timestamp to verify the cache and guarantee that updates are coordinated. Each time the data on Znode changes, the version number connected with it grows.
    There are three different types of Znodes:

- Persistence Znode: These are znodes that continue to function even after the client who created them has been disconnected. Unless otherwise specified, all znodes are persistent by default.
- Ephemeral Znode: Ephemeral znodes are only active while the client is still alive. When the client who produced them disconnects from the ZooKeeper ensemble, the ephemeral Znodes are automatically removed. They have a significant part in the election of the leader.
- Sequential Znode: When znodes are constructed, the ZooKeeper can be asked to append an increasing counter to the path's end. The parent znode's counter is unique. Sequential nodes can be either persistent or ephemeral.

41. ## What is meant by the Replication Tool?

The Replication Tool in Kafka is used for a high-level design to maintain Kafka replicas. Some of the replication tools available are

42. Preferred Replica Leader Election Tool: Partitions are distributed to multiple brokers in a cluster, each copy known as a replica. The preferred replica usually refers to the leader. The brokers distribute the leader role evenly across the cluster for various partitions. Still, an imbalance can occur over time due to failures, planned shutdowns, etc. in such cases, you can use the replication tool to maintain the load balancing by reassigning the preferred replicas and hence, the leaders.
43. Topics tool: Kafka topics tool is responsible for handling all management operations related to topics, which include

- Listing and describing topics
- Creating topics
- Changing topics
- Adding partitions to a topic
- Deleting topics

3. Reassign partitions tool: This tool changes the replicas assigned to a partition. This means adding or removing followers associated with a partition.
4. StateChangeLogMerger tool: This tool is used to collect data from the brokers in a particular cluster, formats it into a central log, and help to troubleshoot issues with state changes. Often, problems may arise with the leader election for a particular partition. This tool can be used to determine what caused the problem.
5. Change topic configuration tool: used to Add new config options, Change existing config options, and Remove config options
6. **How can Kafka be tuned for optimal performance?**
   Tuning for optimal performance involves consideration of two key measures: latency measures, which denote the amount of time taken to process one event, and throughput measures, which refer to how many events can be processed in a specific time. Most systems are optimized for either latency or throughput, while Kafka can balance both. Tuning Kafka for optimal performance involves the following steps:

- Tuning Kafka producers: Data that the producers have to send to brokers is stored in a batch. When the batch is ready, the producer sends it to the broker. For latency and throughput, to tune the producers, two parameters must be taken care of: batch size and linger time. The batch size has to be selected very carefully. If the producer is sending messages all the time, a larger batch size is preferable to maximize throughput. However, if the batch size is chosen to be very large, then it may never get full or take a long time to fill up and, in turn, affect the latency. Batch size will have to be determined, taking into account the nature of the volume of messages sent from the producer. The linger time is

added to create a delay to wait for more records to get filled up in the batch so that larger records are sent. A longer linger time will allow more messages to be sent in one batch, but this could compromise latency. On the other hand, a shorter linger time will result in fewer messages getting sent faster - reduced latency but reduced throughput as well.

- Tuning Kafka broker: Each partition in a topic is associated with a leader, which will further have 0 or more followers. It is important that the leaders are balanced properly and ensure that some nodes are not overworked compared to others.
- Tuning Kafka Consumers: It is recommended that the number of partitions for a topic is equal to the number of consumers so that the consumers can keep up with the producers. In the same consumer group, the partitions are split up among the consumers.

43. **How can all brokers available in a cluster be listed?**
Two ways to get the list of available brokers in an Apache Kafka cluster are as follows:

- Using zookeeper-shell.sh
zookeeper-shell.sh :2181 ls /brokers/ids
Which will give an output like:
WATCHER:: WatchedEvent state:SyncConnected type:None path:null [0, 1, 2, 3]
This indicates that there are four alive brokers - 0,1,2 and 3
- Using zkCli.sh
First, you have to log in to the ZooKeeper client
zkCli.sh -server :2181
Then use the below command to list all the available brokers
ls /brokers/ids
Both the methods used above make use of the ZooKeeper to find out the list of available brokers

44. **What is the Kafka MirrorMaker?**
The Kafka MirrorMaker is a stand-alone tool that allows data to be copied from one Apache Kafka cluster to another. The Kafka MirrorMaker will read data from topics in the original cluster and write the topics to a destination cluster with the same topic name. The source and destination clusters are independent entities and can have different numbers of partitions and varying offset values.

45. **What is meant by Kafka Connect?**
Kafka Connect is a tool provided by Apache Kafka to allow scalable and reliable streaming data to move between Kafka and other systems. It makes it easier to define connectors that are responsible for moving large collections of data in and out of Kafka. Kafka Connect is able to process entire databases as input. It can also collect metrics from application servers into Kafka topics so that this data can be available for Kafka stream processing.

46. **Explain message compression in Apache Kafka.**
In Apache Kafka, producer applications write data to the brokers in JSON format. The data in the JSON format is stored in string form, which can result in several duplicated records getting stored in the Kafka topic. This leads to an increased occupation of disk space. Hence, to reduce this disk space, compression of messages or lingering the data is performed before sending the messages to Kafka. Message compression is done on the producer side, and hence there is no need to make any changes to the configuration of the consumer or the broker.
47. **What is the need for message compression in Apache Kafka?**
Message compression in Kafka does not require any changes in the configuration of the broker or the consumer. It is beneficial for the following reasons:

- Due to reduced size, it reduces the latency in which messages are sent to Kafka.
- Reduced bandwidth allows the producers to send more net messages to the broker.
- When the data is stored in Kafka via cloud platforms, it can reduce the cost in cases where the cloud services are paid.
- Message compression leads to reduced disk load, which will lead to faster read and write requests.

48. **Define consumer lag in Apache Kafka.**
Consumer lag refers to the lag between the Kafka producers and consumers. Consumer groups will have a lag if the data production rate far exceeds the rate at which the data is getting consumed. Consumer lag is the difference between the latest offset and the consumer offset.
49. **What do you know about log compaction in Kafka?**
Log compaction is a method by which Kafka ensures that at least the last known value for each message key within the log of data is retained for a single topic partition. This makes it possible to restore the state after an application crashes or in the event of a system failure. It allows cache reloading once an application restarts during any operational maintenance. Log compaction ensures that any consumer processing the log from the start can view the final state of all records in the original order they were written.
50. **When does Kafka throw a BufferExhaustedException?**
BufferExhaustedException is thrown when the producer cannot allocate memory to a record due to the buffer being too full. The exception is thrown if the producer is in non-blocking mode and the rate of data production exceeds the rate at which data is sent from the buffer for long enough for the allocated buffer to be exhausted.
51. **What are the responsibilities of a Controller Broker in Kafka?**
The main role of the Controller is to manage and coordinate the Kafka cluster, along with the Apache ZooKeeper. Any broker in the cluster can take on the role of the controller. However, once the application starts running, there can be only one controller broker in the

cluster. When the broker starts, it will try to create a Controller node in ZooKeeper. The first broker that creates this controller node becomes the controller.
The controller is responsible for

- creating and deleting topics
- Adding partitions and assigning leaders to the partitions
- Managing the brokers in a cluster - adding new brokers, active broker shutdown, and broker failures
- Leader Election
- Reallocation of partitions.

52. **What causes OutOfMemoryException?**
OutOfMemoryException can occur if the consumers are sending large messages or if there is a spike in the number of messages wherein the consumer is sending messages at a rate faster than the rate of downstream processing. This causes the message queue to fill up, taking up memory.
53. **Explain the graceful shutdown in Kafka.**
Any broker shutdown or failure will automatically be detected by the Apache cluster. In such a case, new leaders will be elected for partitions that were previously handled by that machine. This can occur due to server failure and even if it is intentionally brought down for maintenance or any configuration changes. In cases where the server is intentionally brought down, Kafka supports a graceful mechanism for stopping the server rather than just killing it.
Whenever a server is stopped:

- Kafka ensures that all of its logs are synced onto a disk to avoid needing any log recovery when it is restarted. Since log recovery takes time, this can speed up intentional restarts.
- Any partitions for which the server is the leader will be migrated to the replicas prior to shutting down. This ensures that the leadership transfer is faster, and the time during which each partition is unavailable will be reduced to a few milliseconds.

54. **How can a cluster be expanded in Kafka?**
In order to add a server to a Kafka cluster, it just has to be assigned a unique broker id, and Kafka has to be started on this new server. However, a new server will not automatically be assigned any of the data partitions until a new topic is created. Hence, when a new machine is added to the cluster, it becomes necessary to migrate some existing data to these machines. The partition reassignment tool can be used to move some partitions to the new broker. Kafka will add the new server as a follower of the partition that it is migrating to and allow it to completely replicate the data on that particular partition. When this data is fully replicated, the new server can join the ISR; one of the existing replicas will delete the data that it has with respect to that particular partition.

55. **What is meant by the Kafka schema registry?**
    For both the producers and consumers associated with a Kafka cluster, a Schema Registry is present, which stores Avro schemas. Avro schemas allow the configuration of compatibility settings between the producers and the consumers for seamless serialization and deserialization. Kafka Schema Registry is used to ensure that there is no difference in the schema that is being used by the consumer and the one that is being used by the producer. While using the Confluent schema registry in Kafka, the producers only need to send the schema ID and not the entire schema. The consumer uses the schema ID to look up the corresponding schema in the Schema Registry.
56. **Name the various types of Kafka producer API.**
    There are three types of Kafka producer API available-

- Fire and Forget
- Synchronous producer
- Asynchronous produce

57. **What is the ZooKeeper ensemble?**
    ZooKeeper works as a coordination system for distributed systems and is a distributed system on its own. It follows a simple client-server model, where clients are the machines that make use of the service, and the servers are nodes that provide the service. The collection of ZooKeeper servers forms the ZooKeeper ensemble. Each ZooKeeper server is capable of handling a large number of clients.
58. **What are Znodes?**
    Nodes in a ZooKeeper tree are referred to as znodes. Znodes maintain a structure that contains version numbers for data changes, acl changes, and also timestamps. The version number, along with the timestamp, allows ZooKeeper to validate the cache and ensure that updates are coordinated. The version number associated with Znode increases each time the znode's data changes.
59. **What are the types of Znodes?**
    There are three types of Znodes, namely:
    Persistence Znode: these are the znodes that remain alive even after the client who created that particular znode is disconnected. All znodes are persistent by default unless otherwise specified.
    Ephemeral Znode: Ephemeral znodes remain active only until the client is alive. Ephemeral Znodes get deleted whenever the client that created them gets disconnected from the ZooKeeper ensemble. They play an important role in the leader election.
    Sequential Znode: when znodes are created, it is possible to request the ZooKeeper to add an increasing counter to the end of the path. This counter is unique to the parent znode. Sequential nodes may be persistent or ephemeral.

60. **How can we create Znodes?**
    Znodes are created within the given path.
    Syntax:
    create /path/data
    Flags can be used to specify whether the znode created will be persistent, ephemeral, or sequential.
    create -e /path/data
    creates an ephemeral znode.
    create -s /path/data
    creates a sequential znode.
    All znodes are persistent by default.
61. **Suppose you are sending messages to a Kafka topic using kafkaTemplate. You come across a requirement that states that if a failure occurs while delivering messages to a Kafka topic, you must retry sending the messages on the same partition with the same offset. How can you achieve this using kafkatemplate?**
    If you give the key while delivering the message, it will be stored in the same partition regardless of how many times you send it. The hashed key is used by Kafka to decide which partition needs to be updated.
    The only way to ensure that a failed message has the same offset when retried is to ensure that nothing is put into the topic before retrying it.
62. **Assume your brokers are hosted on AWS EC2. If you're a producer or consumer outside of the Kafka cluster network, you will only be capable of reaching the brokers over their public DNS, not their private DNS. Now, assume your client (producer or consumer) is outside your Kafka cluster's network, and you can only reach the brokers via their public DNS. The private DNS of the brokers hosting the leader partitions, not the public DNS, will be returned by the broker. Unfortunately, since your client is not present on your Kafka cluster's network, they will be unable to resolve the private DNS, resulting in the LEADER NOT AVAILABLE error. How will you resolve this network error?**
    When you first start using Kafka brokers, you might have many listeners. Listeners are just a combination of hostname or IP, port, and protocol.
    Each Kafka broker's server.properties file contains the properties listed below. The important property that will enable you to resolve this network error is advertised.listeners.

- listeners – a list of comma-separated hostnames and ports that Kafka brokers listen to.
- advertised.listeners – a list of comma-separated hostnames and ports that will be returned to clients. Only include hostnames that will be resolved at the client (producer or consumer) level, such as public DNS.
- inter.broker.listener.name – listeners used for internal traffic across brokers. These hostnames do not need to be resolved on the client side, but all of the cluster's brokers

must resolve them.

- listener.security.protocol.map – lists the supported protocols for each listener.

63. **Let's suppose a producer writes records to a Kafka topic at a rate of 10000 messages per second, but the consumer can only read 2500 messages per second. What are the various strategies for expanding your consumer group?**
The solution to this question has two parts: topic partitions and consumer groups. Partitions are used to split a Kafka topic. The producer's message is divided among the topic's partitions based on the message key. You can suppose that the key is chosen in such a way that messages are spread evenly between the partitions.
Consumer groups are a method of grouping consumers together to maximize a consumer application's throughput. Each consumer in a consumer group holds on to a topic partition. If the Kafka topic has four partitions and the consumer group has four consumers, each consumer will read from a single partition. If there are six partitions and four consumers, the data will be read in parallel from only four partitions. As a result, maintaining a 1-to-1 mapping of partition to the consumer in the consumer group is preferable.
Now, you can do two things to increase processing on the consumer side:

- You can increase the topic's partition count (say from existing 1 to 4).
- You can build a Kafka consumer group with four consumer instances tied to it.
This would enable the consumers to read data from the topic in parallel, allowing it to expand from 2500 to 10000 messages per second.

64. ## What is Kafka’s producer acknowledgment? What are the various types of acknowledgment settings that Kafka provides?
A broker sends an ack or acknowledgment to the producer to verify the reception of the message. Ack level is a configuration parameter in the Producer that specifies how many acknowledgments the producer must receive from the leader before a request is considered successful. The following types of acknowledgment are available:

- acks=0
In this setting, the producer does not wait for the broker's acknowledgment. There is no way to know if the broker has received the record.
- acks=1
In this situation, the leader logs the record to its local log file and answers without waiting for all of its followers to acknowledge it. The message can only be lost in this instance if the leader fails shortly after accepting the record but before the followers have copied it; otherwise, the record would be lost.

- acks=all
  A set leader in this situation waits for all in-sync replica sets to acknowledge the record. As long as one replica is alive, the record will not be lost, and the best possible guarantee will be provided. However, because a leader must wait for all followers to acknowledge before replying, the throughput is significantly lower.

65. **How do you get Kafka to perform in a FIFO manner?**
    Kafka organizes messages into topics, which are then divided into partitions. The partition is an immutable list of ordered messages that is updated regularly. A message in the partition is uniquely recognized by a sequential number called offset. FIFO behavior is possible only within the partitions. Following the methods below will help you achieve FIFO behavior:

- To begin, we first set the enable the auto-commit property to be false:
  Set enable.auto.commit=false
- We should not call the consumer.commitSync(); method after the messages have been processed.
- Then we may "subscribe" to the topic and ensure that the consumer system's register is updated.
- You should use Listener consumerRebalance, and call a consumer inside a listener. seek(topicPartition, offset).
- The offset related to the message should be kept together with the processed message once it has been processed.

66. **Explain Kafka's message delivery semantics.**
    Kafka offers three message delivery semantics: At most once, At least once, and Exactly once, ensuring different trade-offs between message delivery and duplication.
67. **Explain the role of log segments in Kafka.**
    Log segments are files that store Kafka messages. They are immutable and are used to manage disk space by performing log segment rolling and deletion.
68. **What is the purpose of the offset in Kafka?**
    The offset is a unique identifier of a record within a partition. It denotes the position of the consumer in the partition. Kafka maintains this offset per partition, per consumer group, allowing each consumer group to read from a different position in the partition. This enables Kafka to provide both queue and publish-subscribe messaging models.
69. **How would you secure a Kafka cluster?**
    Top candidates would use multiple layers of security and strategies such as:

- SSL/TLS for encryption of data in transit
- SASL/SCRAM for authentication

- A Kerberos integration
- Network policies for controlling access to the Kafka cluster
- ACLs (Access Control Lists) for authorizing actions by users or groups on specific topics

70. **Explain the concept of Kafka MirrorMaker.**
    Kafka MirrorMaker is a tool used for cross-cluster data replication. It enables data mirroring between two Kafka clusters, which is particularly useful for disaster recovery and geo-replication scenarios.
    MirrorMaker works by using consumer and producer configurations to pull data from a source cluster and push it to a destination cluster.
71. **What are ISR in Kafka?**
    ISR (short for In-Sync Replicas) are replicas of a Kafka partition that are fully in sync with the leader. They're critical for ensuring data durability and consistency. If a leader fails, one of the ISRs can become the new leader.
72. **What authentication mechanisms can you use in Kafka?**
    Kafka supports:

- **SSL/TLS** for encrypting data and optionally authenticating clients using certificates
- **SASL** (Simple Authentication and Security Layer) which supports mechanisms like **GSSAPI** (Kerberos), **PLAIN**, and **SCRAM** to secure Kafka brokers against unauthorized access
- Integration with **enterprise authentication systems** like LDAP

73. **Describe an instance where Kafka might lose data and how you would prevent it.**
    A good response will mention cases such as unclean leader elections, broker failures, or configuration errors that lead to data loss.
    Candidates should explain how they'd configure Kafka's replication factors, min.insync.replicas, and acknowledgment settings to prevent data loss. They should also mention they'd do regular backups and set up consistent monitoring to prevent issues.
74. **What is `linger.ms` in Kafka producers?**

- Definition: `linger.ms` is a producer configuration that specifies the time (in milliseconds) the producer waits before sending a batch of messages.
- Behavior:
  - If the batch is full (`batch.size` reached), it is sent immediately.
  - If the batch is not full, the producer waits for the `linger.ms` time before sending the batch, hoping more records will arrive.
- Purpose:
  - To improve throughput by batching more records into a single request.
  - Reduces the number of network calls but may slightly increase latency.

- Default Value: `0`, meaning no waiting and the producer sends records as soon as possible.
  Example Scenario: If `linger.ms = 10` and `batch.size` isn't reached, the producer will wait 10ms before sending the batch, potentially grouping more messages together.

75. **How does Kafka manage backpressure?**
    Kafka handles backpressure by controlling the flow of data between producers, brokers, and consumers through these mechanisms:

- Producer-Side:
  - Buffering: Producers buffer records up to `buffer.memory`. If the buffer is full, the producer blocks or throws an exception (based on `max.block.ms`).
  - Batching: Producers optimize sending data in batches (`batch.size`) to handle high-throughput workloads efficiently.
- Broker-Side:
  - Replication Quotas: Kafka enforces quotas for replication to ensure brokers aren't overwhelmed.
  - I/O Throttling: Limits disk and network I/O rates to maintain cluster stability.
- Consumer-Side:
  - Pause and Resume: Consumers can pause fetching records if they can't process fast enough, avoiding memory overload.
  - Fetch Min/Max Bytes: Controls how much data is fetched at a time to prevent excessive resource usage.

76. **CommitSync() vs CommitAsync() in Kafka consumers**

Aspect	commitSync()	commitAsync()
Type	Synchronous	Asynchronous
Blocking	Blocks until the broker acknowledges the commit	Does not block; continues processing
Reliability	Highly reliable; throws exception on failure	Less reliable; errors may be ignored
Performance	Slower due to waiting for acknowledgment	Faster due to non-blocking behavior
Use Case	Critical systems (e.g., financial transactions)	High-throughput systems (e.g., analytics)

Aspect	commitSync()	commitAsync()
Error Handling	Direct exception handling	Handle via a callback function