Out-of-order Kafka Messages

Kafka users often encounter is the ordering of the messages, and one of the scenarios on the consumer side is the need to re-order the received messages during processing.

There are various reasons the messages can be out-of-order. First of all, Kafka only guarantees message ordering within a partition, not across partitions.

This places a burden on the producers and consumers to follow certain Kafka design patterns to ensure ordering.

For example, the ability to partition data by key and one consumer per partition. There are cases where we want to have flexibility, or cases where we cannot follow these design patterns.

One of the solutions in these cases is to let the consumer re-order the messages during processing.

Latency and throughput:

Latency and throughput are both important considerations when designing a Kafka streaming application. In order to optimize both, it's important to consider the following:

Choice of Kafka topics:

The choice of Kafka topic can have a significant impact on latency and throughput. For example, a topic with fewer partitions may be able to provide lower latency, while a topic with more partitions may be able to provide higher throughput.

Use of batching and compression:

Batching and compression can help to reduce network overhead and improve throughput. However, they may also increase latency, so it's important to carefully consider the trade-offs.

Configuration of the Kafka client and server:

The configuration of the Kafka client(Producer & Consumer) and server can also have a significant impact on latency and throughput. For example, the use of a faster network or higher bandwidth can help to improve throughput, while the use of a faster CPU or more memory can help to reduce latency.

Latency and throughput:

batch.size: This property specifies the number of messages that should be batched together before being sent to Kafka. Increasing this value can help to improve throughput, but may increase latency.

compression.type: This property specifies the type of compression that should be used for messages sent to Kafka. Compression can help to improve throughput, but may increase latency.
Ex: snappy, gzip etc

linger.ms: This property specifies the amount of time that the producer should wait before sending a batch of messages to Kafka. Increasing this value can help to improve throughput by allowing more messages to be batched together, but may increase latency.

max.request.size: This property specifies the maximum size of a single message that can be sent to Kafka. This can impact both latency and throughput, as larger messages may take longer to send and may require more network bandwidth.

compression.type

1.gzip: Offers a good balance between compression ratio and CPU usage. It provides moderate compression ratios with relatively low CPU overhead. However, decompression may introduce slightly higher latency compared to other compression types.

2.snappy: Known for its fast compression and decompression speeds with moderate compression ratios. Snappy compression consumes less CPU compared to gzip, which can lead to lower latency. It's a good choice if minimizing latency is a priority.

3.lz4: Provides the fastest compression and decompression speeds among the available compression types in Kafka. It offers low CPU usage and low latency, making it suitable for scenarios where minimizing latency is critical. However, it may sacrifice some compression ratio compared to gzip or snappy.

Data and state persistence:

Kafka provides durable, fault-tolerant storage for streams of data, which can be used to provide data and state persistence for streaming applications. When designing a Kafka streaming application, it's important to consider the following:

Use of stateful processing: Kafka streams can be used to perform stateful processing, such as aggregations or joins, which can help to provide data and state persistence.

Choice of storage mechanism: Kafka can be used to store stateful data, but other storage mechanisms, such as a database or a distributed cache, may also be used to provide additional scalability or performance.

Handling of failure scenarios: When using stateful processing and data persistence, it's important to consider failure scenarios, such as node failures or network partitions. It's important to design the application to handle these scenarios in a graceful manner, such as by using replication or failover mechanisms.

Data and state persistence:

state.dir: This property specifies the directory where Kafka streams should store state information. This can be used to provide fault-tolerance and scalability for stateful processing.

cleanup.policy: This property specifies the retention policy for Kafka topics. This can impact the amount of data that is persisted, and can be used to manage disk space usage.

num.standby.replicas: This property specifies the number of standby replicas that should be maintained for each partition. Standby replicas can be used to provide fault-tolerance for stateful processing.

```
bin/kafka-topics.sh --create --topic my-topic --partitions 6
--replication-factor 3 --config min.insync.replicas=2
--config num.replica.fetchers=3 --config num.standby.replicas=2
–bootstrap-server localhost:9092
```

--replication-factor option is set to 3 to create three replicas of each partition, and the --config num.standby.replicas option is set to 2 to create two standby replicas for each partition.

num.replica.fetchers is a configuration parameter in Apache Kafka that specifies the number of replica fetcher threads that are used by the broker to copy data from leader replicas to follower and standby replicas.

The leader replica, which is the replica that is currently serving read and write requests for the partition, is included in the replication factor of 3.

The other two replicas are follower replicas that are kept in sync with the leader replica and can serve read requests if needed.

The two standby replicas are kept in sync with the leader replica, but are not used for serving read or write requests unless the leader replica fails.

In Confluent, standby replicas can be used by consumers to consume messages, but this feature requires the use of a specific client library called Confluent's Kafka Consumer.

By default, Kafka clients consume messages only from the leader replicas, as they are the only replicas that can serve read requests. However, Confluent's Kafka Consumer library can be configured to consume messages from the follower replicas as well, which can provide additional benefits such as improved read performance and reduced latency.

To enable consuming messages from follower replicas in Confluent's Kafka Consumer library, you can set the replica.reads configuration parameter to "true". This parameter specifies whether the consumer should read from follower replicas as well as leader replicas. When set to "true", the consumer will periodically fetch data from follower replicas and use it to serve read requests.

In Apache Kafka, the interval at which follower replicas sync with the leader replica is controlled by the **replica.fetch.interval.ms** configuration parameter.

This parameter specifies the maximum amount of time that a follower replica can wait before fetching new data from the leader replica. The default value for this parameter is **1 second**. This means that by default, follower replicas will fetch new data from the leader replica at least once every second.

It's important to note that the actual frequency at which follower replicas sync with the leader replica can also be affected by other factors, such as the amount of data that needs to be replicated, network latency, and the availability of resources on the Kafka brokers. Additionally, the **replica.fetch.max.bytes** configuration parameter can also impact the frequency of fetches, as it specifies the maximum amount of data that can be fetched in a single request. If the amount of data that needs to be fetched exceeds this limit, multiple fetch requests may be needed, which can increase the time between fetches.

There are several reasons why a replica may become out of sync with the leader replica. For example, wrong configuration of GC, network issues or broker failures can cause replication to be delayed or interrupted. Additionally, if the min.insync.replicas parameter is set too high, replicas may struggle to keep up with the leader replica and fall behind.

When a replica falls out of sync with the leader replica, it will attempt to catch up by fetching any missing data from the leader replica. If the replica is unable to catch up within a configurable timeout period (specified by the **replica.lag.time.max.ms** parameter : default 10sec), it will be considered "offline" and removed from the in-sync replicas set for the partition.

Data sources:

Kafka can be used to consume data from a wide variety of data sources. When designing a Kafka streaming application, it's important to consider the following:

Frequency of updates: The frequency of updates to the data source can impact the design of the application. For example, if the data is updated frequently, it may be more efficient to perform incremental updates rather than full updates.

Volume of data: The volume of data can impact the performance of the application. For example, if the volume of data is very large, it may be necessary to use partitioning and parallel processing to ensure that the data can be processed in a timely manner.

Format of the data: The format of the data can impact the ease of processing and serialization. For example, if the data is in a complex format, such as XML, it may be more difficult to process and serialize than if it were in a simpler format, such as JSON.

Data sources:

num.partitions: This property specifies the number of partitions that should be used for a Kafka topic. Increasing the number of partitions can improve throughput and enable parallel processing.

fetch.min.bytes: This property specifies the minimum amount of data that a consumer should fetch from Kafka in a single request. This can impact both latency and throughput, as larger values may result in longer wait times for data.

fetch.max.bytes: This property specifies the maximum amount of data that a consumer can fetch from Kafka in a single request. This can impact both latency and throughput, as larger values may require more network bandwidth.

External data lookups:

Kafka can be used to perform external data lookups, such as performing a database query or calling an external API. When designing a Kafka streaming application, it's important to consider the following:

Frequency of lookups: The frequency of lookups can impact the performance of the application. For example, if the lookups are very frequent, it may be necessary to use caching or other optimization techniques to reduce the number of lookups.

Complexity of lookup logic: The complexity of the lookup logic can impact the design of the application. For example, if the lookup logic is very complex, it may be necessary to perform the lookups in a separate processing step or on a separate node.

Performance of the external system: The performance of the external system can impact the overall performance of the application. For example, if the external system is slow or has limited capacity, it may impact the throughput and Latency.

External data lookups:

max.block.ms: This property specifies the maximum amount of time that a consumer should wait for new data from Kafka. This can impact latency, as longer wait times can result in longer processing times.

max.poll.records: This property specifies the maximum number of records that a consumer can fetch in a single poll request. This can impact both latency and throughput, as larger values may result in longer processing times.

enable.auto.commit: This property specifies whether or not the consumer should automatically commit its offset position to Kafka. Enabling this can simplify processing, but may impact the accuracy of the offset position if the consumer fails.

Data formats:

key.serializer and value.serializer: These properties specify the serializer classes that should be used to convert data objects to byte arrays for sending to Kafka. Examples of commonly used serializers include Avro, JSON, and Protobuf.

key.deserializer and value.deserializer: These properties specify the deserializer classes that should be used to convert byte arrays received from Kafka back into data objects. These should correspond to the serializer classes used by the producer.

Data serialization:

schema.registry.url: This property specifies the URL of the schema registry used for Avro data serialization. The schema registry is used to manage and store Avro schemas, which are required to serialize and deserialize data.

auto.register.schemas: This property specifies whether or not new Avro schemas should be automatically registered with the schema registry when they are encountered for the first time.

default.key.serde and default.value.serde: These properties specify the Serde classes that should be used to serialize and deserialize data objects. Serde classes are used to handle both serialization and deserialization, and can be used for multiple data formats (such as JSON, Avro, or Protobuf).

Level of parallelism:

num.stream.threads: This property specifies the number of threads that should be used for processing Kafka streams. Increasing this value can enable more parallel processing, but may require more resources.

partition.assignment.strategy: This property specifies the partition assignment strategy to be used by Kafka consumers. The default strategy is to use a round-robin approach, but other strategies are available that can enable more fine-grained control over partition assignment.

concurrency: This property specifies the level of concurrency that should be used by a Kafka consumer or producer. For example, a higher concurrency value for a consumer can enable it to process multiple records concurrently, which can improve throughput.

In Apache Kafka, the partition.assignment.strategy configuration parameter determines how partitions are assigned to consumers in a consumer group.

By default, Kafka uses the **Range assignment strategy,** which assigns contiguous partitions to each consumer in the group based on the range of partition IDs.

For example, if there are 10 partitions in a topic and 3 consumers in a group, the first consumer will be assigned partitions 0-3, the second consumer will be assigned partitions 4-6, and the third consumer will be assigned partitions 7-9.

However, Kafka also provides several other partition assignment strategies that can be used instead of or in addition to the default Range strategy. These include:

RoundRobin: Assigns partitions to consumers in a round-robin fashion, such that each consumer receives one partition at a time until all partitions have been assigned.

Sticky: Assigns partitions to consumers such that each consumer retains ownership of the same partitions across rebalances, as long as they remain active and in the group.

Custom: Allows users to define their own partition assignment logic by implementing the PartitionAssignor interface.

Kafka troubleshooting and best practices

Out-of-order events:

To prevent out-of-order events, we can use the partitioning feature in Kafka. This ensures that messages with the same key are sent to the same partition, so they are processed in the correct order.

We can also use the "max.poll.interval.ms" property in the consumer configuration to set a time limit for processing each message. If a message takes longer than this time to process, it will be considered a failure and retried.

Message processing semantics:

To ensure reliable message delivery, you can use the "acks" property in the producer configuration to set the number of acknowledgments required from the broker.

A value of "all" ensures that all brokers receive the message before the producer receives an acknowledgement.

We can also use the "isolation.level" property in the consumer configuration to set the read isolation level. This determines whether the consumer reads messages that have been committed or uncommitted.

Best practices:

To ensure high performance and reliability, you can use the following best practices:

Set appropriate replication factors and retention policies
Monitor Kafka metrics, including disk usage, lag, and error rates.

Use compression to reduce message size and improve network throughput.

Properly configure the producer and consumer, including buffer sizes and concurrency levels.

Allocate enough resources, including CPU, memory, and disk space

Case studies:

Some companies that use Kafka include:

LinkedIn, which uses Kafka for activity data processing and messaging, processing over 1.5 trillion messages per day.
Netflix, which uses Kafka for logging and metrics collection, collecting over 4 billion events per day.

Uber, which uses Kafka for real-time streaming and data processing, processing over 100 billion events per day.

Alerting and monitoring:

To monitor Kafka, you can use the built-in metrics provided by Kafka, including broker, producer, and consumer metrics. You can also use third-party monitoring tools, such as Prometheus and Grafana. To set up alerts, you can use tools like Nagios or Zabbix, which can send notifications when certain metrics exceed predefined thresholds.

Useful Kafka metrics:
Some useful Kafka metrics to monitor include:

Broker CPU and memory usage
Disk usage and I/O rates
Network throughput and latency
Topic and partition lag
Producer and consumer message rates

Kafka runtime management using JMX:

Monitoring broker metrics:

We can use JMX to monitor various broker metrics, such as disk usage, CPU usage, and network I/O. You can connect to the JMX port (usually 9999) using a JMX client like JConsole or JVisualVM. Some useful broker MBeans to monitor include:

kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
kafka.server:type=KafkaController,name=ActiveControllerCount

Configuring the producer and consumer:

We can use JMX to configure the producer and consumer properties at runtime. For example, you can use the following MBeans to set the producer buffer size and compression type:

kafka.producer:type=ProducerTopicMetrics,name=BufferPoolWaitTimeMs
kafka.producer:type=ProducerTopicMetrics,name=BufferPoolAvailableBytes
kafka.producer:type=ProducerTopicMetrics,name=CompressionRate
Similarly, you can use the following MBeans to set the consumer fetch size and maximum poll interval:

kafka.consumer:type=ConsumerFetcherManager,name=BytesPerSecond
kafka.consumer:type=consumer-fetch-manager-metrics,name=MaxPollIntervalMs
kafka.consumer:type=consumer-fetch-manager-metrics,name=FetchRequestsPerSec

Troubleshooting issues:

We can use JMX to troubleshoot various issues in Kafka. For example, you can use the following MBeans to monitor the leader election process:

kafka.controller:type=KafkaController,name=LeaderElectionRateAndTimeMs
kafka.controller:type=KafkaController,name=UncleanLeaderElectionsPerSec

We can also use JMX to monitor the log cleanup process, by monitoring the "LogCleaner" MBean.

Flow control in Kafka is the mechanism used to regulate the rate at which data is produced and consumed. It is used to prevent producer from overwhelming the broker with too many messages or the consumer from processing messages too quickly, which can result in increased memory usage, network congestion, and potential data loss.

In Kafka, flow control is implemented using a combination of buffer sizes, quotas, and backpressure.

When a producer sends messages to a broker, it stores them in a buffer. If the buffer becomes full, the producer can either block until more space becomes available, or it can use a callback to handle the overflow.

Similarly, when a consumer fetches messages from a broker, it stores them in a local buffer. If the buffer becomes too large, the consumer can either slow down the rate at which it fetches messages, or it can use backpressure to request the broker to stop sending messages until it catches up.

Kafka provides several configuration parameters to control flow control, including:

buffer.memory: The amount of memory the producer and consumer use to buffer data.

batch.size: The number of messages sent together in a batch by the producer.

linger.ms: The amount of time the producer waits before sending a batch of messages.

max.request.size: The maximum size of a single message that the producer can send to the broker.

fetch.min.bytes: The minimum amount of data the consumer fetches in a single request.

fetch.max.bytes: The maximum amount of data the consumer fetches in a single request.

By tuning these parameters, we can control the rate at which data is produced and consumed, ensuring that your Kafka cluster operates efficiently and reliably.

Some potential solutions for the Kafka errors mentioned:

Connection Errors: To solve connection issues, you can try the following steps:

Check the network configuration to ensure that the client and broker are on the same network or that any firewall or security settings are not blocking the connection.

Verify that the Kafka broker is running and available by checking its status and logs.

Check the client configuration to ensure that the correct broker address is specified and that any authentication settings are correct.

Broker Errors: To solve broker issues, you can try the following steps:

Check the broker logs for any error messages and take appropriate corrective action based on the cause of the error. Restart the broker and ensure that any dependencies are also running correctly.

Replace the broker with a new one and ensure that it is configured correctly and is running properly.

Producer Errors: To solve producer issues, you can try the following steps:

Check the producer configuration settings and verify that they are correct, including the topic and broker addresses. Verify that the data being published is in the correct format and is not exceeding any size limits.

Check the producer logs for any error messages and take appropriate corrective action based on the cause of the error.

Consumer Errors: To solve consumer issues, you can try the following steps:

Check the consumer configuration settings and verify that they are correct, including the topic and broker addresses.

Verify that the consumer is subscribed to the correct topic and partition and that it has the necessary permissions to read data.

Check the consumer logs for any error messages and take appropriate corrective action based on the cause of the error.

Partition Errors: To solve partition issues, you can try the following steps:

Verify that the partition exists and is correctly configured, including replication factor and number of replicas.

Check the partition logs for any error messages and take appropriate corrective action based on the cause of the error. If necessary, reassign or recreate the partition to resolve the issue.

Serialization Errors: To solve serialization issues, you can try the following steps:

Verify that the data being produced or consumed is in the correct format and matches the specified schema.

Check the serialization and deserialization configuration settings to ensure that they are correct.

Check the producer or consumer logs for any error messages related to serialization or deserialization and take appropriate corrective action based on the cause of the error.

Resource Errors: To solve resource issues, you can try the following steps:

Monitor resource usage in the Kafka cluster and identify any bottlenecks or areas of high usage.

Increase the resources available to the Kafka components, such as CPU, memory, or disk space.

Optimize Kafka configuration settings to reduce resource usage and improve performance.

The number of brokers, partitions, and replication factor in a Kafka cluster should be chosen based on several factors, such as the expected volume of data, the number of producers and consumers, and the desired level of fault tolerance and scalability. Here is some more detailed information on how to decide on these parameters:

Number of Brokers:
The number of brokers in a Kafka cluster affects the overall performance, scalability, and fault tolerance of the system. Here are some general guidelines for choosing the number of brokers:

Start with a small number of brokers and scale up as needed. Aim for at least three brokers to achieve fault tolerance and high availability.

Consider factors such as network bandwidth, disk I/O, and CPU usage when determining the number of brokers.

Number of Partitions:

The number of partitions in a Kafka topic affects the overall throughput and parallelism of the system. Here are some general guidelines for choosing the number of partitions:

Aim for a large enough number of partitions to achieve high throughput and parallelism.

Consider the number of producers and consumers and their processing capacity when determining the number of partitions.

Avoid creating too many partitions, as this can lead to increased overhead and decreased performance.

Replication Factor:

The replication factor in a Kafka topic affects the fault tolerance and durability of the system. Here are some general guidelines for choosing the replication factor:

Aim for a replication factor of at least three to achieve high availability and fault tolerance.

Consider factors such as the expected data volume and the number of consumers when determining the replication factor. Avoid setting the replication factor too high, as this can lead to increased overhead and decreased performance.

In general, it is important to monitor the Kafka cluster and adjust the number of brokers, partitions, and replication factor as needed based on changing conditions and requirements.

Kafka is an open-source distributed streaming platform that is widely used for real-time data processing, messaging, and analytics. Kafka provides a scalable, fault-tolerant, and highly available platform for handling large volumes of data in real-time. However, security is a critical aspect of any distributed system, and Kafka is no exception. In this overview, we will explore some of the key aspects of securing Kafka.

Wire Encryption using SSL:
Encryption is a critical aspect of securing data in transit. Kafka provides support for wire encryption using SSL (Secure Sockets Layer) protocol. SSL is a widely used encryption protocol that provides a secure communication channel between a client and a server. When SSL is enabled, Kafka clients and brokers can communicate securely over the network by encrypting the data in transit.

To enable SSL encryption in Kafka, you need to generate SSL certificates and configure SSL settings on both the client and the broker side. SSL can provide end-to-end encryption between Kafka clients and brokers, ensuring that data is protected from unauthorized access and interception.

Kerberos SASL for Authentication:

Authentication is the process of verifying the identity of a client or a user. Kafka supports authentication using Kerberos SASL (Simple Authentication and Security Layer). Kerberos is a widely used authentication protocol that provides strong authentication and secure communication between clients and servers.

When Kerberos SASL is enabled, Kafka clients and brokers can authenticate each other using Kerberos tickets. The clients and brokers exchange tickets to establish a secure communication channel. This ensures that only authorized clients can access Kafka brokers and data.

Understanding ACL and Authorization:

Authorization is the process of controlling access to resources based on user permissions. Kafka provides support for authorization using ACL (Access Control List). ACL allows you to control access to Kafka topics, partitions, and other resources based on user permissions.

ACL provides fine-grained control over who can read, write, and execute operations on Kafka resources. You can set ACL rules on Kafka brokers and clients to control access to resources. This ensures that only authorized users can access Kafka resources and data.

Understanding Zookeeper Authentication:

Zookeeper is a distributed coordination service that is used by Kafka to store metadata, configuration, and other essential data. Zookeeper is a critical component of Kafka, and securing Zookeeper is essential for securing Kafka.

Zookeeper supports authentication using Kerberos SASL and SSL encryption. When Zookeeper authentication is enabled, clients must authenticate themselves using a Kerberos ticket or SSL certificate to access Zookeeper. This ensures that only authorized clients can access Zookeeper data, and the Zookeeper service remains secure.

Kafka Security using ACL and authorization, you will need access to a Kafka cluster and a Zookeeper server. Here are the detailed steps:

Install Kafka and Zookeeper:
You will need to install Kafka and Zookeeper on your local machine or on a remote server. Follow the Kafka and Zookeeper installation instructions from the official documentation.

Configure Kafka Broker:
Edit the Kafka server properties file (server.properties) and add the following lines to enable ACL:

authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer

This will enable SimpleAclAuthorizer, which is used for access control.

Create Kafka Topics:
Create a new Kafka topic using the following command:

bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic my_topic

This will create a new topic called my_topic.

Configure ACL:
Configure ACL rules for the topic using the following command:

bin/kafka-acls.sh --authorizer-properties –bootstrap-server localhost:9092
--add --allow-principal User:producer --operation Write --topic my_topic

This will produce a message to the my_topic topic using the producer user.
If the ACL rule is correctly configured, the message will be produced successfully.

Test ACL:

Test the ACL rule by producing a message to the topic using the following command:

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my_topic --producer-property security.protocol=PLAINTEXT --producer-property "sasl.mechanism=PLAIN" --producer-property "sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username=producer password=producer-secret;"

This will produce a message to the my_topic topic using the producer user. If the ACL rule is correctly configured, the message will be produced successfully.

Configure ACL for Consumers:

Configure ACL rules for the consumers using the following command:
bin/kafka-acls.sh --authorizer-properties --bootstrap-server localhost:9092
 --add --allow-principal User:consumer --operation Read --topic my_topic

This will allow the consumer user to read from the my_topic topic.

Test ACL for Consumers:
Test the ACL rule by consuming a message from the topic using the following command:

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my_topic --consumer-property security.protocol=PLAINTEXT --consumer-property "sasl.mechanism=PLAIN" --consumer-property "sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username=consumer password=consumer-secret;"