

Message Queue CKafka

General References

Product Documentation



Copyright Notice

©2013-2019 Tencent Cloud. All rights reserved.

Copyright in this document is exclusively owned by Tencent Cloud. You must not reproduce, modify, copy or distribute in any way, in whole or in part, the contents of this document without Tencent Cloud's the prior written consent.

Trademark Notice



All trademarks associated with Tencent Cloud and its services are owned by Tencent Cloud Computing (Beijing) Company Limited and its affiliated companies. Trademarks of third parties referred to in this document are owned by their respective proprietors.

Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

General References

Conducting Production and Consumption Pressure Testing on CKafka

Configuration Guide for Common Parameters in CKafka

Connecting to Legacy Self-Built Kafka

Suggestions for CKafka Version Selection

CKafka Data Reliability Description

General References

Conducting Production and Consumption Pressure Testing on CKafka

Last updated : 2021-09-16 16:01:53

Testing Tool

The open-source script of the Kafka client can be used for Kafka producer and consumer performance testing. Test results are displayed mainly based on the size of messages sent per second (MB/second) and the number of messages sent per second (records/second).

- Kafka producer test script: `$KAFKA_HOME/bin/kafka-producer-perf-test.sh`
- Kafka consumer test script: `$KAFKA_HOME/bin/kafka-consumer-perf-test.sh`

Testing Command

Note :

The `ckafka vip:vport` in the following sample commands should be replaced by the actual IP and port assigned for your instance.

Sample command for production testing:

```
bin/kafka-producer-perf-test.sh
--topic test
--num-records 123
--record-size 1000
--producer-props bootstrap.servers= ckafka vip : port
--throughput 20000
```

Sample command for consumption testing:

```
bin/kafka-consumer-perf-test.sh
--topic test
--new-consumer
--fetch-size 10000
```

```
--messages 1000  
--broker-list bootstrap.servers=ckafka vip : port
```

Suggestions

- We recommend you create three or more partitions to increase throughput. This is because there must be at least three CKafka cluster nodes at the backend. If only one partition is created, it will be distributed in a single broker, which will affect CKafka performance.
- As messages in each CKafka partition are ordered, the production performance will be affected if there are too many partitions. We recommend you create up to six partitions.
- It is necessary to simulate concurrency with multiple clients to ensure the testing effect. We recommend you use multiple test servers as the pressure test clients (producers) and start multiple pressure test programs on each test server to increase concurrency. To avoid the high load of test servers, you are also recommended to start one producer every second rather than all producers simultaneously.

Configuration Guide for Common Parameters in CKafka

Last updated : 2021-04-16 11:38:16

Broker Configuration Parameter Description

The following are the configurations of a CKafka Broker for your reference:

```
# Maximum message length in bytes.
message.max.bytes=1000012

# Whether to allow automatic creation of topics. The default value is false. Currently, topics can be created in the console or through the Tencent Cloud API.
auto.create.topics.enable=false

# Whether to allow topic deletion by calling the API.
delete.topic.enable=true

# The maximum request length allowed for a Broker is 16 MB.
socket.request.max.bytes=16777216

# Each IP can establish up to 5,000 connections with a Broker.
max.connections.per.ip=5000

# Offset retention period. The default value is 7 days.
offsets.retention.minutes=10080

# Everyone is allowed to access when there is no ACL configuration.
allow.everyone.if.no.acl.found=true

# The log segment size is 1 GB.
log.segment.bytes=1073741824

# The log rolling check interval is 5 minutes. If the retention period is set to less than 5 minutes, it may still take 5 minutes to clear logs.
log.retention.check.interval.ms=300000
```

Note :

For configurations not listed here, see the [open-source Kafka default configurations](#).

Topic Configuration Parameter Description

1. Number of partitions

From the producer's point of view, writes to different partitions are completely in parallel; from the consumer's point of view, the number of concurrencies depends entirely on the number of partitions (if there are more consumers than partitions, there will definitely be idle consumers). It is important to select an appropriate number of partitions to fully play the performance of the CKafka instance.

The number of partitions should be determined based on the throughput of production and consumption, ideally through the following formula:

Note :

$$\text{Num} = \max(T/PT, T/CT) = T/\min(PT, CT)$$

Num represents the number of partitions, T the target throughput, PT the maximum production throughput by the producer to a single partition, and CT the maximum consumption throughput by the consumer from a single partition.

The number of partitions is equal to T/PT or T/CT, whichever is larger.

In practice, the actual PT is determined by batch size, compression algorithm, acknowledgement mechanism, number of replicas, and so on, while the actual CT is subject to business logic, which varies according to the actual conditions.

We recommend that the number of partitions be greater than or equal to that of consumers to achieve maximum concurrency. For example, if there are 5 consumers, there should be 5 or more partitions.

However, having too many partitions will lower production throughput and increase time consumed by elections and thus need to be avoided. See the following for reference:

- A single partition can implement sequential writes of messages.
- A single partition can only be consumed by a single consumer process in the same consumer group.
- A single consumer process can consume multiple partitions simultaneously, so partition limits the concurrency of consumers.
- The more partitions there are, the longer it takes to elect a leader upon failure.
- Offset can be down to the partition level. The more partitions there are, the more time the offset query consumes.
- The number of partitions can be dynamically increased but not reduced. However, an increase will result in message rebalance.

2. Number of replicas

At present, the number of replicas must be at least 2 to ensure availability. To ensure high reliability, we recommend maintaining at least 3 replicas.

Note :

The number of replicas will affect the production/consumption traffic; for example, if there are 3 replicas, the actual traffic will be 3 times the production traffic.

3. Log retention period

The `log.retention.ms` configuration of a topic is set through the retention period of the instance in the console.

4. Other topic-level configurations

```
# Maximum message length at the topic level.
max.message.bytes=1000012

# Messages in the 0.10.2 version are in the V1 format.
message.format.version=0.10.2-IV0

# Replica not in ISR can be selected as a leader; in this case, availability is higher than reliability, and there is a risk of data loss.
unclean.leader.election.enable=true

# Minimum number of replicas for producer requests submitted by ISR. If the number of replicas in synchronization is below this value, the server will no longer accept write requests where request.required.acks is set to -1 or all.
min.insync.replicas=1
```

Producer Configuration Guide

The following describes common parameter settings for the Producer client. We recommend adjusting them based on your actual business scenarios:

```
# The producer will attempt to bundle and send the messages sent to the same partition by the business to the Broker. The batch.size parameter sets the maximum size of the bundle, which is 16 KB by default. If batch.size is too small, the throughput will drop; if it is too large, too much memory will be used.
batch.size=16384

# The following describes the 3 ACK mechanisms supported by a Kafka producer:
# -1 or all: the Broker responds to the producer and continues to send the next message or next batch of messages only after the leader receives the data and syncs it to followers in all ISRs. This configuration provides the highest data reliability, as messages will never be lost as long as one synced replica survives. Note: this configuration cannot ensure that replicas will be returned after the dat
```


a is written to all of them. It can be used together with the `min.insync.replicas` parameter at the topic level.

0: the producer continues to send the next message or next batch of messages without waiting for acknowledgement of synchronization completion from the Broker. This configuration provides the highest production performance but lowest data reliability (data loss may occur when the server fails. If the leader is dead but the producer is unaware of that, the Broker cannot receive messages).

1: the producer sends the next message or next batch of messages after it receives an acknowledgement that the leader has successfully received the data. This configuration is a balance between production throughput and data reliability (messages may be lost if the leader is dead but has not been replicated yet).

If users do not configure this, the default value will be 1. Users can customize this according to their business conditions.

```
acks=1
```

Control the maximum time a production request waits in the Broker for replica synchronization to meet conditions set through acks.

```
timeout.ms=30000
```

Configure the memory that the producer uses to cache messages to be sent to the Broker. Users must adjust this according to the total memory size of the process where the producer resides.

```
buffer.memory=33554432
```

If messages are produced faster than they are sent by the sender thread to the Broker, the memory configured by `buffer.memory` will be used up and thus block the send operation by the producer. This parameter sets the maximum blocking time.

```
max.block.ms=60000
```

Set the time to send the scheduled message, so that more messages can be sent in batches. This parameter is 0 by default, indicating immediate send. When the messages to be sent reach the size set by `batch.size`, the request will be sent immediately regardless of whether the time set by `linger.ms` is reached.

```
linger.ms=0
```

*# Maximum size of the request packet that the producer can send, which defaults to 1 MB. **Note:** this value cannot exceed the maximum packet size of 16 MB configured for the Broker.*

```
max.request.size=1048576
```

Compression format configuration. Currently, version 0.9 and earlier do not support compression, and version 0.10 and later do not support gzip compression.

```
compression.type=[none, snappy, lz4]
```

Timeout period for the client to send a request to the Broker, which cannot be smaller than the value of `replica.lag.time.max.ms` configured for the Broker. The

```
current value is 10,000 ms.
request.timeout.ms=30000

# Maximum number of unacknowledged requests that the client can send on each connection. If this parameter is greater than 1 and retries is greater than 0, data may be out of order. If strict ordering is necessary, set this value to 1.
max.in.flight.requests.per.connection=5

# Number of retries upon request error. It is recommended that you set the parameter to a value greater than 0 to enable retries and ensure to the greatest extent that messages do not get lost.
retries=0

# Retry interval upon request failure.
retry.backoff.ms=100
```

Consumer Configuration Guide

The following describes common parameter settings for the Consumer client. We recommend adjusting them based on your actual business scenarios:

```
# Whether to sync the offset to the Broker after a message is consumed, so the latest offset can be obtained from the Broker when the consumer fails.
auto.commit.enable=true

# Interval for the automatic submission of offset when auto.commit.enable=true is configured. We recommend setting this value to at least 1,000.
auto.commit.interval.ms=5000

# Mode to initialize the offset when no offset is configured for the Broker (such as upon initial consumption or when the offset expired for more than 7 days), or mode to reset the offset when error OFFSET_OUT_OF_RANGE occurs.
# earliest: reset to the minimum offset in the partition.
# latest: reset to the maximum offset in the partition. This is the default value
.
# none: throw an OffsetOutOfRangeException exception without resetting the offset
.
auto.offset.reset=latest

# Identify the consumer group to which the consumer belongs.
group.id=""

# Consumer timeout period when the Kafka consumer groups are used. If the Broker
```

does **not** receive the heartbeat **of** the consumer **within** this period, the consumer will be considered **to** have failed **and** the Broker will initiate rebalance. Currently, this **value** must be configured **in** the Broker **between** **6000** (**value of group.min.session.timeout.ms**) **and** **300000** (**value of group.max.session.timeout.ms**).

```
session.timeout.ms=10000
```

Interval at which the consumer sends a heartbeat **when** the Kafka consumer **groups** are used. This **value** must be smaller than the **session.timeout.ms** **value**, **and is** recommended **to** be smaller than one third **of** it.

```
heartbeat.interval.ms=3000
```

Maximum **interval** allowed **for** calling the poll again **when** the Kafka consumer **groups** are used. **If** poll **is not called within** this **time** period, the consumer will be considered **to** have failed **and** the Broker will re-initiate rebalance **to** assign the partitions **to** other consumers.

```
max.poll.interval.ms=300000
```

Minimum data size returned **by** a **fetch** request. The **default value is** **1** B, indicating that the request can be returned **as soon as** possible. Increasing this **value** will increase throughput **and** latency.

```
fetch.min.bytes=1
```

Maximum data size returned **by** a **fetch** request. The **default value is** **50** MB.

```
fetch.max.bytes=52428800
```

Fetch request wait **time**.

```
fetch.max.wait.ms=500
```

Maximum data size returned **by each partition in** a **fetch** request. The **default value is** **1** MB.

```
max.partition.fetch.bytes=1048576
```

Number **of** records returned **in** one poll **call**.

```
max.poll.records=500
```

Client request timeout period. **If no** response **is** received **after** this **time** elapses, the request will **time out and** fail.

```
request.timeout.ms=305000
```

Connecting to Legacy Self-Built Kafka

Last updated : 2022-03-28 17:47:17

CKafka is compatible with the Producer and Consumer APIs of Apache Kafka 0.9 and above (currently, directly purchasable versions include v0.10.2, v1.1.1, v2.4.1, and v2.8.1). To connect to an on-premises Kafka of an earlier version (such as v0.8), partial rewriting of APIs is needed. This document compares the Producer and Consumer APIs of Kafka 0.8 and its earlier versions, and describes how to rewrite the APIs.

Kafka Producer

Overview

In Kafka 0.8.1, the Producer API is rewritten. This Producer client version is officially recommended because it provides better performance and more features. The community will maintain the new version of the Producer API (referred to as the New Producer API).

Comparison between new producer API and old producer API

- New Producer API Demo

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:4242");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(0), Integer.toString(0)));
producer.close();
```

- Old Producer API Demo

```
Properties props = new Properties();
props.put("metadata.broker.list", "broker1:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
```

```
props.put("partitioner.class", "example.producer.SimplePartitioner");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<String, String> producer = new Producer<String, String>(config);
KeyedMessage<String, String> data = new KeyedMessage<String, String>("page_visits", ip, msg);
producer.send(data);
producer.close();
```

As shown in the previous code, the basic usage of the new and old versions are the same, except for some parameter settings. This means the cost of API partial rewriting is not high.

Compatibility description

Producer API 0.8.x can be connected to CKafka successfully without partial rewriting. We recommend using the New Kafka Producer API.

Kafka Consumer

Overview

The open-source Apache Kafka 0.8 provides two types of the Consumer API:

- High Level Consumer API (blocking configuration details)
- Simple Consumer API (support for parameter configuration adjustment)

Kafka 0.9.x has introduced the New Consumer API that inherits the features of the two types of the old consumer API (v0.8) and reduces the load on ZooKeeper.

The following describes how to transform the old consumer API (v0.8) to the new consumer API (v0.9).

Comparison between new consumer API and old consumer API

Old consumer API (v0.8)

- **High Level Consumer API** ([Demo](#))

The High Level Consumer API can meet general requirements if you care only about data, except for message offset. This API, built on the consumer group logic, blocks the offset management, and supports Broker fault handling and Consumer load balancing. It allows developers to get started with the Consumer client quickly.

Consider the following when you use the High Level Consumer API:

- If the number of consumer threads is greater than the number of partitions, certain consumer threads cannot obtain data.
- If the number of partitions is greater than the number of threads, certain threads consume more than one partition.
- The changes in partitions and consumers will affect rebalancing.

- **Low Level Consumer API** ([Demo](#))

The Low Level Consumer API is recommended if you need message offset and features like repeated consumption or skip read, or if you want to consume specific partitions and ensure more consumption semantics. But in this case, you need to handle offsets and Broker exceptions.

When using the Low Level Consumer API, you need to:

- Track and maintain the offset and control the consumption progress.
- Find the leader of partitions for the topic, and deal with partition changes.

New consumer API (v0.9)

Kafka 0.9.x has introduced the New Consumer API that inherits the features of the two types of Old Consumer API while providing consumer coordination (High Level API) and lower-level access to customize consumption policies. The New Consumer also simplifies the consumer client and introduces a central coordinator to solve the herd effect and split-brain problems resulting from the separate connections to ZooKeeper, and to reduce the load on ZooKeeper.

Advantages:

- Introduces coordinators
The current version of High Level Consumer has the herd effect and split-brain problems. Placing failure detection and rebalancing logics into a highly available central coordinator solves both problems while greatly reducing the load on the ZooKeeper.
- Allows you to assign partitions
To keep certain states of each local partition unchanged, you need to keep partition mappings unchanged. Some other scenarios are designed to associate the Consumer with the region-dependent Broker.
- Allows you to manage offsets
You can manage offsets as needed to implement repeated consumption, skipped consumption, and other semantics.
- Triggers callbacks after rebalancing based on your specifications
- Provides non-blocking Consumer API

Comparison between new consumer API and old consumer API

Type	Version	Auto Offset Storage	Auto Offset Management	Auto Exception Handling	Auto Rebalance Handling	Auto Leader Search	Pros and Cons
High Level Consumer	Before 0.9	Supported	Not supported	Supported	Supported	Supported	Herd effect and split b

Type	Version	Auto Offset Storage	Auto Offset Management	Auto Exception Handling	Auto Rebalance Handling	Auto Leader Search	Pros and Cons
Simple Consumer	Before 0.9	Not supported	Supported	Not supported	Not supported	Not supported	Various exceptions need to be handled
New Consumer	After 0.9	Supported	Supported	Supported	Supported	Supported	Mature and recommended for the current version

Transforming old consumer to new consumer

- New Consumer

```
//The main configuration difference is that the ZooKeeper parameters are replaced.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
//Compared with old consumers, new consumers are easier to create.
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
ConsumerRecords<String, String> records = consumer.poll(100);
for (ConsumerRecord<String, String> record : records)
System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());}
```

- Old Consumer (High Level)

```
//Old consumers require ZooKeeper
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
```

```
props.put("group.id", "test");
props.put("auto.commit.enable", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("auto.offset.reset", "smallest");
ConsumerConfig config = new ConsumerConfig(props);
//Require connector creation
ConsumerConnector connector = Consumer.createJavaConsumerConnector(config);
//Create a message stream
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put("foo", 1);
Map<String, List<KafkaStream<byte[], byte[]>>> streams =
connector.createMessageStreams(topicCountMap);
//Obtain data
KafkaStream<byte[], byte[]> stream = streams.get("foo").get(0);
ConsumerIterator<byte[], byte[]> iterator = stream.iterator();
MessageAndMetadata<byte[], byte[]> msg = null;
while (iterator.hasNext()) {
msg = iterator.next();
System.out.println("//
" group " + props.get("group.id") + //
", partition " + msg.partition() + ", " + //
new String(msg.message()));
}
```

Comparing with the old consumer, the new consumer has simpler coding and uses Kafka addresses instead of ZooKeeper parameters. In addition, the new consumer has parameter settings for interactions with coordinators, in which the default settings are suitable for general use.

Compatibility description

Both CKafka and the new version of Kafka in the open-source community support the rewritten new consumer API, which blocks the interaction between the consumer client and ZooKeeper (ZooKeeper is not exposed to users any longer). The new consumer solves the herd effect and split brain problems resulting from direct interaction with ZooKeeper, and integrates the features of the old consumer, thus making the consumption more reliable.

Suggestions for CKafka Version Selection

Last updated : 2021-08-18 15:30:18

This document describes the compatibility of CKafka with open-source Kafka and helps you select a CKafka version that is more suitable for your business according to your business needs.

Overview

Open-Source Kafka has about 20 versions ranging from v0.7.x to v2.8.x. From the perspective of message queue, it can be divided into three stages: v0.x, v1.x, and v2.x. At present, Tencent Cloud provides four corresponding versions for these three community development stages: v0.10, v1.1, v2.4, and v2.8, which basically cover commonly used Kafka versions.

Among them, the two major versions v1.x and v2.x mainly contain optimizations and improvements of Kafka Streams but don't introduce many major features in terms of the message engine (v2.x has great improvements of the transaction feature though). Kafka Streams is greatly improved on v2.x. Therefore, if you need to use these features, please select v2.x at least.

Compatibility Description

CKafka is perfectly compatible with open-source Kafka with full backward compatibility with lower versions. For example, if you build Kafka v0.10 on your own, you can select CKafka v0.10, v1.1.1, or v2.4.1 in the cloud, but if you build Kafka on a higher version, we recommend you not select a lower version (because it is uncertain whether your business uses features of higher versions).

The following describes the compatibility:

CKafka Version	Compatible Community Versions	Compatibility
0.10.2	≤ 0.10.x	100%
1.1.1	≤ 1.1.x	100%
2.4.1	≤ 2.4.x	100%
2.8.1	≤ 2.8.x	100%

Note for CKafka v2.4.1

When CKafka launched v2.4, the stable branch in the community was v2.4.1. Later, the community launched a development branch v2.4.2. After some repairs were merged, it was positioned as v2.4.2. Then, the community deleted v2.4.2, so there was no v2.4.2 in the community eventually. Therefore, the v2.4.2 previously displayed by CKafka is aligned with the current v2.4.1.

Suggestions for CKafka Version Selection

- If you migrate your self-built Kafka to the cloud, we recommend you select the corresponding major version. For example, if your self-built Kafka is on v1.1.0, please select CKafka v1.1.
- If the corresponding version cannot be found in the cloud, we recommend you select a higher version. For example, if your self-built Kafka is on v1.0.0, we recommend you use CKafka v1.1.1, and if it is on v0.11.x, we also recommend you use CKafka v1.1.1 (because each version of Broker is backward compatible).

CKafka Data Reliability Description

Last updated : 2022-03-28 17:47:17

This document describes the factors that affect the reliability of CKafka from the perspectives of the producer, the service (CKafka), and the consumer, respectively, and provides corresponding solutions.

What should I do if data is lost on the producer?

Causes of data loss

When the producer sends data to CKafka, the data may be lost due to network jitters, and CKafka will not receive the data. This may be caused by following reasons:

- The network load is high or the disk is busy, and the producer does not have a retry mechanism.
- The purchased disk capacity is exceeded. For example, if the disk capacity of an instance is 9,000 GB, and it is not expanded promptly after being used up, data cannot be written to CKafka.
- Sudden or continuously increased peak traffic exceeds the purchased peak throughput. For example, if the peak throughput of the instance is 100 MB/s, but it is not scaled up promptly after the peak throughput is exceeded for a long period of time, data writes to CKafka will become slower. In this case, if the producer has a queuing timeout mechanism in place, data cannot be written to CKafka.

Solution

- Enable the retry mechanism on the producer for important data.
- For disk usage, set up monitors and [alarm policies](#) as preventive measures when configuring the instance. When the disk capacity is used up, upgrade the instance timely in the console. Upgrading non-exclusive CKafka instances will not interrupt the service, which also supports expanding the disk capacity alone. You can also shorten the message retention period to reduce disk usage.
- In order to minimize the loss of messages at the producer side, you can fine-tune the size of the buffer using `buffer.memory` and `batch.size` (in bytes). For the buffer, bigger is not always better. If the producer is down for some reason, the more data in the buffer, the more garbage to be recycled, and the slower the recovery.
Pay close attention to the number of messages produced by the producer and average message size (through the rich set of monitoring metrics available in CKafka).

- Configure ACK for the producer.

When the producer sends data to the leader, the data reliability level can be set using the

`request.required.acks` and `min.insync.replicas` parameters.

- When `acks = 1` (default value), the producer's leader in ISR has successfully received a data entry, and the next data entry can be sent. If the leader is down, as the data may have not been synced to its followers, the data will be lost.
- When `acks = 0`, the producer sends the next message without waiting for acknowledgment from the broker. In this case, data transfer efficiency is the highest, but data reliability is the lowest.

Note

When the producer is configured with `acks = 0`, if the current instance is throttled, in order for the server to provide services normally, the server will actively close the connection with the client.

- When `acks = -1` or `all`, the producer needs to wait for acknowledgment of message receipt from all the followers in ISR before sending the next message, which leads to highest reliability.
Even if ACK is configured as above, there is no guarantee that data will never be lost. For example, when there is only one leader in ISR (the number of members in ISR may increase or decrease in certain circumstances, and in some cases, only one leader is left), `acks` will be 1. Therefore, you also need to use the `min.insync.replicas` parameter (which can be configured in the CKafka console > Topic Management > Advanced Configuration). It represents the minimum number of replicas in ISR, which is 1 by default and takes effect when and only when `acks = -1` or `all`.

Recommended parameter values

These parameter values are for reference only, and actual values depend on the actual conditions of your business.

- Retry mechanism: `message.send.max.retries=3;retry.backoff.ms=10000;`
- Guarantee for high reliability: `request.required.acks=-1;min.insync.replicas=2;`
- Guarantee for high performance: `request.required.acks=0;`
- Reliability + performance: `request.required.acks=1;`

What should I do if data is lost on the server (CKafka)?

Causes of data loss

- The partition's leader is down before the backup of followers is completed. Even if a new leader is elected, data will be lost because it has not been backed up yet.
- Open-source Kafka features async storage to disk, that is, data is first stored in PageCache. If the broker disconnects, restarts, or fails, the data stored in PageCache will be lost because it has not been stored to the disk yet.

- Stored data may be lost due to disk failure.

Solution

- Open-source Kafka has a multi-replica feature. You are recommended to use replicas to ensure data integrity. Data will be lost only if multiple replicas and multiple brokers fail at the same time, so data reliability is higher than that in the single-replica case. Therefore, CKafka requires at least 2 replicas for a topic and supports configuring 3 replicas.
- More reasonable values are configured for the `log.flush.interval.messages` and `log.flush.interval.ms` parameters in CKafka for data flushing.
- In CKafka, the disk is specially designed to ensure that data reliability will not be compromised even if the disk is partially damaged.

Recommended parameter values

A replica which is not in syncing status can be elected as a leader:

```
unclean.leader.election.enable=false // Disabled
```

What should I do if data is lost on the consumer?

Causes of data loss

- The offset is committed before data is consumed. If the consumer is down in the process but the offset has been updated, the consumer will miss a data entry, and the consumer group will have to reset the offset in order to retrieve it.
- The consumption speed differs significantly from the production speed, but the message retention period is too short; therefore, the message will be deleted upon expiration before it is consumed.

Solution

- Configure the `auto.commit.enable` parameter appropriately. When it is `true`, commit is performed automatically. You are recommended to use scheduled commit so as to avoid frequent commit offset.
- Monitor the consumer and correctly adjust the data retention period. Monitor the consumption offset and the number of messages that are not consumed, and configure an alarm to prevent messages from being deleted upon expiration due to slow consumption.

Troubleshooting data loss

Printing partition and offset information locally for troubleshooting

Below is the code for printing information:

```
Future<RecordMetadata> future = producer.send(new ProducerRecord<>(topic, message
Key, messageStr));
RecordMetadata recordMetadata = future.get();
log.info("partition: {}", recordMetadata.partition());
log.info("offset: {}", recordMetadata.offset());
```

- If the partition and offset can be printed out, the currently sent message has been correctly saved on the server. At this time, you can use the message query tool to query the information of the relevant offset.
- If the partition and offset cannot be printed out, the message has not been saved on the server, and the client needs to retry.