Red Hat AMQ Streams 2.6

Kafka configuration tuning

Use Kafka configuration properties to optimize the streaming of data

Last Updated: 2023-12-06

Red Hat AMQ Streams 2.6 Kafka configuration tuning

Use Kafka configuration properties to optimize the streaming of data

Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Fine-tune the operation of Kafka brokers, producers, and consumers using Kafka configuration properties.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE 3

CHAPTER 1. KAFKA TUNING OVERVIEW 4
1.1. MAPPING PROPERTIES AND VALUES 4
1.2. TOOLS THAT HELP WITH TUNING 4

CHAPTER 2. MANAGED BROKER CONFIGURATION 5

CHAPTER 3. KAFKA BROKER CONFIGURATION TUNING 6
3.1. BASIC BROKER CONFIGURATION 6
3.2. REPLICATING TOPICS FOR HIGH AVAILABILITY 6
3.3. INTERNAL TOPIC SETTINGS FOR TRANSACTIONS AND COMMITS 7
3.4. IMPROVING REQUEST HANDLING THROUGHPUT BY INCREASING I/O THREADS 7
3.5. INCREASING BANDWIDTH FOR HIGH LATENCY CONNECTIONS 9
3.6. MANAGING KAFKA LOGS WITH DELETE AND COMPACT POLICIES 9
3.7. MANAGING EFFICIENT DISK UTILIZATION FOR COMPACTION 13
3.8. HANDLING LARGE MESSAGE SIZES 13
3.9. CONTROLLING THE LOG FLUSH OF MESSAGE DATA 15
3.10. PARTITION REBALANCING FOR AVAILABILITY 16
3.11. UNCLEAN LEADER ELECTION 17
3.12. AVOIDING UNNECESSARY CONSUMER GROUP REBALANCES 17

CHAPTER 4. KAFKA CONSUMER CONFIGURATION TUNING 18
4.1. BASIC CONSUMER CONFIGURATION 18
4.2. SCALING DATA CONSUMPTION USING CONSUMER GROUPS 18
4.3. MESSAGE ORDERING GUARANTEES 19
4.4. OPTIMIZING CONSUMERS FOR THROUGHPUT AND LATENCY 19
4.5. AVOIDING DATA LOSS OR DUPLICATION WHEN COMMITTING OFFSETS 20
4.5.1. Controlling transactional messages 21
4.6. RECOVERING FROM FAILURE TO AVOID DATA LOSS 21
4.7. MANAGING OFFSET POLICY 22
4.8. MINIMIZING THE IMPACT OF REBALANCES 22

CHAPTER 5. KAFKA PRODUCER CONFIGURATION TUNING 24
5.1. BASIC PRODUCER CONFIGURATION 24
5.2. DATA DURABILITY 24
5.3. ORDERED DELIVERY 26
5.4. RELIABILITY GUARANTEES 26
5.5. OPTIMIZING PRODUCERS FOR THROUGHPUT AND LATENCY 27

APPENDIX A. USING YOUR SUBSCRIPTION 30
Accessing Your Account 30
Activating a Subscription 30
Downloading Zip and Tar Files 30
Installing packages with DNF 30

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. KAFKA TUNING OVERVIEW

Use configuration properties to optimize the performance of Kafka brokers, producers and consumers. You can specify configuration properties for AMQ Streams on OCP and RHEL.

A minimum set of configuration properties is required, but you can add or adjust properties to change how producers and consumers interact with Kafka brokers. For example, you can tune latency and throughput of messages so that clients can respond to data in real time.

You might start by analyzing metrics to gauge where to make your initial configurations, then make incremental changes and further comparisons of metrics until you have the configuration you need.

For more information about Apache Kafka configuration properties, see the Apache Kafka documentation.

1.1. MAPPING PROPERTIES AND VALUES

How you specify configuration properties depends on the type of deployment. If you deployed AMQ Streams on OCP, you can use the **Kafka** resource to add configuration for Kafka brokers through the **config** property. With AMQ Streams on RHEL, you add the configuration to a properties file as environment variables.

When you add **config** properties to custom resources, you use a colon (':') to map the property and value.

Example configuration in a custom resource

```
num.partitions:1
```

When you add the properties as environment variables, you use an equal sign ('=') to map the property and value.

Example configuration as an environment variable

```
num.partitions=1
```

1.2. TOOLS THAT HELP WITH TUNING

The following tools help with Kafka tuning:

- Cruise Control generates optimization proposals that you can use to assess and implement a cluster rebalance
- Kafka Static Quota plugin sets limits on brokers
- Rack configuration spreads broker partitions across racks and allows consumers to fetch data from the nearest replica

For more information on these tools, see the following guides:

- Configuring AMQ Streams on OpenShift
- Using AMQ Streams on RHEL

CHAPTER 2. MANAGED BROKER CONFIGURATION

When you deploy AMQ Streams on OpenShift, you specify broker configuration through the **config** property of the **Kafka** custom resource. However, certain broker configuration options are managed directly by AMQ Streams.

As such, if you are using AMQ Streams on OpenShift, you cannot configure the following options:

- **broker.id** to specify the ID of the Kafka broker
- **log.dirs** directories for log data
- **zookeeper.connect** configuration to connect Kafka with ZooKeeper
- **listeners** to expose the Kafka cluster to clients
- **authorization** mechanisms to allow or decline actions executed by users
- **authentication** mechanisms to prove the identity of users requiring access to Kafka

Broker IDs start from 0 (zero) and correspond to the number of broker replicas. Log directories are mounted to **/var/lib/kafka/data/kafka-log*IDX*** based on the **spec.kafka.storage** configuration in the **Kafka** custom resource. *IDX* is the Kafka broker pod index.

For a list of exclusions, see the **KafkaClusterSpec** schema reference.

These exclusions don’t apply when using AMQ Streams on RHEL. In this case, you need to add these properties in your basic broker configuration to identify your brokers and provide secure access.

Example broker configuration for AMQ Streams on RHEL

```
# ...
broker.id = 1
log.dirs = /var/lib/kafka
zookeeper.connect = zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181
listeners = internal-1://:9092
authorizer.class.name = kafka.security.auth.SimpleAclAuthorizer
ssl.truststore.location = /path/to/truststore.jks
ssl.truststore.password = 123456
ssl.client.auth = required
# ...
```

Additional resources

- Configuring Kafka on OCP
- Configuring Kafka on RHEL

CHAPTER 3. KAFKA BROKER CONFIGURATION TUNING

Use configuration properties to optimize the performance of Kafka brokers. You can use standard Kafka broker configuration options, except for properties managed directly by AMQ Streams.

3.1. BASIC BROKER CONFIGURATION

A typical broker configuration will include settings for properties related to topics, threads and logs.

Basic broker configuration properties

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...
```

3.2. REPLICATING TOPICS FOR HIGH AVAILABILITY

Basic topic properties set the default number of partitions and replication factor for topics, which will apply to topics that are created without these properties being explicitly set, including when topics are created automatically.

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
# ...
```

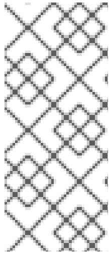
For high availability environments, it is advisable to increase the replication factor to at least 3 for topics and set the minimum number of in-sync replicas required to 1 less than the replication factor.

The **auto.create.topics.enable** property is enabled by default so that topics that do not already exist are created automatically when needed by producers and consumers. If you are using automatic topic creation, you can set the default number of partitions for topics using **num.partitions**. Generally, however, this property is disabled so that more control is provided over topics through explicit topic creation.

For data durability, you should also set **min.insync.replicas** in your *topic* configuration and message delivery acknowledgments using **acks=all** in your *producer* configuration.

Use **replica.fetch.max.bytes** to set the maximum size, in bytes, of messages fetched by each follower that replicates the leader partition. Change this value according to the average message size and throughput. When considering the total memory allocation required for read/write buffering, the memory available must also be able to accommodate the maximum replicated message size when multiplied by all followers.

The **delete.topic.enable** property is enabled by default to allow topics to be deleted. In a production environment, you should disable this property to avoid accidental topic deletion, resulting in data loss. You can, however, temporarily enable it and delete topics and then disable it again.



NOTE

When running AMQ Streams on OpenShift, the Topic Operator can provide operator-style topic management. You can use the **KafkaTopic** resource to create topics. For topics created using the **KafkaTopic** resource, the replication factor is set using **spec.replicas**. If **delete.topic.enable** is enabled, you can also delete topics using the **KafkaTopic** resource.

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

3.3. INTERNAL TOPIC SETTINGS FOR TRANSACTIONS AND COMMITS

If you are using transactions to enable atomic writes to partitions from producers, the state of the transactions is stored in the internal **__transaction_state** topic. By default, the brokers are configured with a replication factor of 3 and a minimum of 2 in-sync replicas for this topic, which means that a minimum of three brokers are required in your Kafka cluster.

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
# ...
```

Similarly, the internal **__consumer_offsets** topic, which stores consumer state, has default settings for the number of partitions and replication factor.

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

Do not reduce these settings in production. You can increase the settings in a *production* environment. As an exception, you might want to reduce the settings in a single-broker *test* environment.

3.4. IMPROVING REQUEST HANDLING THROUGHPUT BY INCREASING I/O THREADS

Network threads handle requests to the Kafka cluster, such as produce and fetch requests from client applications. Produce requests are placed in a request queue. Responses are placed in a response queue.

The number of network threads per listener should reflect the replication factor and the levels of activity from client producers and consumers interacting with the Kafka cluster. If you are going to have a lot of requests, you can increase the number of threads, using the amount of time threads are idle to determine when to add more threads.

To reduce congestion and regulate the request traffic, you can limit the number of requests allowed in the request queue. When the request queue is full, all incoming traffic is blocked.

I/O threads pick up requests from the request queue to process them. Adding more threads can improve throughput, but the number of CPU cores and disk bandwidth imposes a practical upper limit. At a minimum, the number of I/O threads should equal the number of storage volumes.

```
# ...
num.network.threads=3 (1)
queued.max.requests=500 (2)
num.io.threads=8 (3)
num.recovery.threads.per.data.dir=4 (4)
# ...
```

(1) The number of network threads for the Kafka cluster.

(2) The number of requests allowed in the request queue.

(3) The number of I/O threads for a Kafka broker.

(4) The number of threads used for log loading at startup and flushing at shutdown. Try setting to a value of at least the number of cores.

Configuration updates to the thread pools for all brokers might occur dynamically at the cluster level. These updates are restricted to between half the current size and twice the current size.

TIP

The following Kafka broker metrics can help with working out the number of threads required:

- **kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent** provides metrics on the average time network threads are idle as a percentage.
- **kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent** provides metrics on the average time I/O threads are idle as a percentage.

If there is 0% idle time, all resources are in use, which means that adding more threads might be beneficial. When idle time goes below 30%, performance may start to suffer.

If threads are slow or limited due to the number of disks, you can try increasing the size of the buffers for network requests to improve throughput:

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

And also increase the maximum number of bytes Kafka can receive:

```
# ...
socket.request.max.bytes=104857600
# ...
```

3.5. INCREASING BANDWIDTH FOR HIGH LATENCY CONNECTIONS

Kafka batches data to achieve reasonable throughput over high-latency connections from Kafka to clients, such as connections between datacenters. However, if high latency is a problem, you can increase the size of the buffers for sending and receiving messages.

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

You can estimate the optimal size of your buffers using a *bandwidth-delay product* calculation, which multiplies the maximum bandwidth of the link (in bytes/s) with the round-trip delay (in seconds) to give an estimate of how large a buffer is required to sustain maximum throughput.

3.6. MANAGING KAFKA LOGS WITH DELETE AND COMPACT POLICIES

Kafka relies on logs to store message data. A log consists of a series of segments, where each segment is associated with offset-based and timestamp-based indexes. New messages are written to an *active* segment and are never subsequently modified. When serving fetch requests from consumers, the segments are read. Periodically, the active segment is *rolled* to become read-only, and a new active segment is created to replace it. There is only one active segment per topic-partition per broker. Older segments are retained until they become eligible for deletion.

Configuration at the broker level determines the maximum size in bytes of a log segment and the time in milliseconds before an active segment is rolled:

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

These settings can be overridden at the topic level using **segment.bytes** and **segment.ms**. The choice to lower or raise these values depends on the policy for segment deletion. A larger size means the active segment contains more messages and is rolled less often. Segments also become eligible for deletion less frequently.

In Kafka, log cleanup policies determine how log data is managed. In most cases, you won’t need to change the default configuration at the cluster level, which specifies the **delete** cleanup policy and enables the log cleaner used by the **compact** cleanup policy:

```
# ...
log.cleanup.policy=delete
log.cleaner.enable=true
# ...
```

Delete cleanup policy

Delete cleanup policy is the default cluster-wide policy for all topics. The policy is applied to topics that do not have a specific topic-level policy configured. Kafka removes older segments based on time-based or size-based log retention limits.

Compact cleanup policy

Compact cleanup policy is generally configured as a topic-level policy (**cleanup.policy=compact**). Kafka's log cleaner applies compaction on specific topics, retaining only the most recent value for a key in the topic. You can also configure topics to use both policies (**cleanup.policy=compact,delete**).

Setting up retention limits for the delete policy

Delete cleanup policy corresponds to managing logs with data retention. The policy is suitable when data does not need to be retained forever. You can establish time-based or size-based log retention and cleanup policies to keep logs bounded.

When log retention policies are employed, non-active log segments are removed when retention limits are reached. Deletion of old segments helps to prevent exceeding disk capacity.

For time-based log retention, you set a retention period based on hours, minutes, or milliseconds:

```
# ...
log.retention.ms=1680000
# ...
```

The retention period is based on the time messages were appended to the segment. Kafka uses the timestamp of the latest message within a segment to determine if that segment has expired or not. The milliseconds configuration has priority over minutes, which has priority over hours. The minutes and milliseconds configurations are null by default, but the three options provide a substantial level of control over the data you wish to retain. Preference should be given to the milliseconds configuration, as it is the only one of the three properties that is dynamically updateable.

If **log.retention.ms** is set to -1, no time limit is applied to log retention, and all logs are retained. However, this setting is not generally recommended as it can lead to issues with full disks that are difficult to rectify.

For size-based log retention, you specify a minimum log size (in bytes):

```
# ...
log.retention.bytes=1073741824
# ...
```

This means that Kafka will ensure there is always at least the specified amount of log data available.

For example, if you set **log.retention.bytes** to 1000 and **log.segment.bytes** to 300, Kafka will keep 4 segments plus the active segment, ensuring a minimum of 1000 bytes are available. When the active segment becomes full and a new segment is created, the oldest segment is deleted. At this point, the size on disk may exceed the specified 1000 bytes, potentially ranging between 1200 and 1500 bytes (excluding index files).

A potential issue with using a log size is that it does not take into account the time messages were appended to a segment. You can use time-based and size-based log retention for your cleanup policy to get the balance you need. Whichever threshold is reached first triggers the cleanup.

To add a time delay before a segment file is deleted from the system, you can use **log.segment.delete.delay.ms** at the broker level for all topics:

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

Or configure **file.delete.delay.ms** at the topic level.

You set the frequency at which the log is checked for cleanup in milliseconds:

```
# ...
log.retention.check.interval.ms=300000
# ...
```

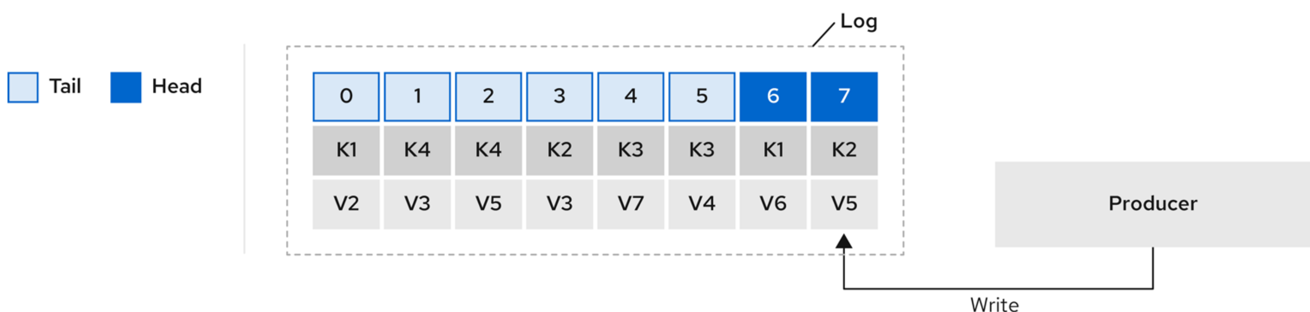
Adjust the log retention check interval in relation to the log retention settings. Smaller retention sizes might require more frequent checks. The frequency of cleanup should be often enough to manage the disk space but not so often it affects performance on a broker.

Retaining the most recent messages using compact policy

When you enable log compaction for a topic by setting **cleanup.policy=compact**, Kafka uses the log cleaner as a background thread to perform the compaction. The compact policy guarantees that the most recent message for each message key is retained, effectively cleaning up older versions of records. The policy is suitable when message values are changeable, and you want to retain the latest update.

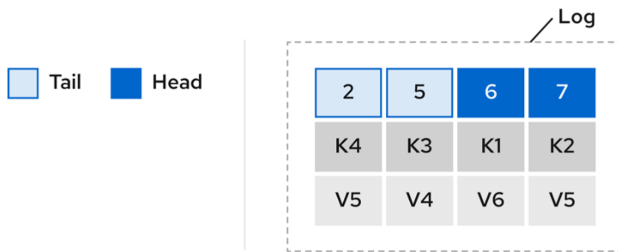
If a cleanup policy is set for log compaction, the *head* of the log operates as a standard Kafka log, with writes for new messages appended in order. In the *tail* of a compacted log, where the log cleaner operates, records are deleted if another record with the same key occurs later in the log. Messages with null values are also deleted. To use compaction, you must have keys to identify related messages because Kafka guarantees that the latest messages for each key will be retained, but it does not guarantee that the whole compacted log will not contain duplicates.

Figure 3.1. Log showing key value writes with offset positions before compaction



Using keys to identify messages, Kafka compaction keeps the latest message (with the highest offset) that is present in the log tail for a specific message key, eventually discarding earlier messages that have the same key. The message in its latest state is always available, and any out-of-date records of that particular message are eventually removed when the log cleaner runs. You can restore a message back to a previous state. Records retain their original offsets even when surrounding records get deleted. Consequently, the tail can have non-contiguous offsets. When consuming an offset that's no longer available in the tail, the record with the next higher offset is found.

Figure 3.2. Log after compaction



If appropriate, you can add a delay to the compaction process:

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

The deleted data retention period gives time to notice the data is gone before it is irretrievably deleted.

To delete all messages related to a specific key, a producer can send a *tombstone* message. A tombstone has a null value and acts as a marker to inform consumers that the corresponding message for that key has been deleted. After some time, only the tombstone marker is retained. Assuming new messages continue to come in, the marker is retained for a duration specified by **log.cleaner.delete.retention.ms** to allow consumers enough time to recognize the deletion.

You can also set a time in milliseconds to put the cleaner on standby if there are no logs to clean:

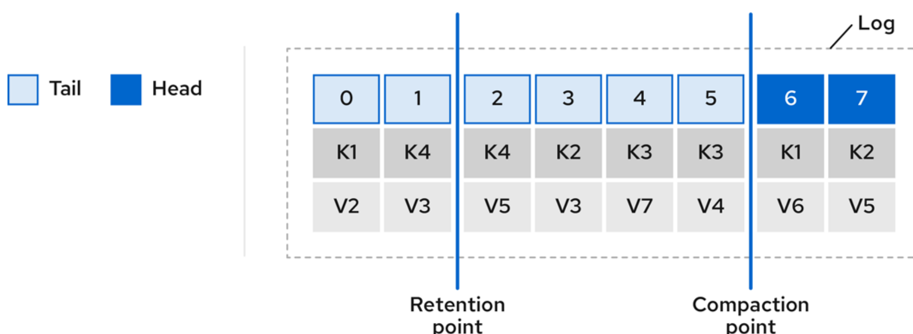
```
# ...
log.cleaner.backoff.ms=15000
# ...
```

Using combined compact and delete policies

If you choose only a compact policy, your log can still become arbitrarily large. In such cases, you can set the cleanup policy for a topic to compact and delete logs. Kafka applies log compaction, removing older versions of records and retaining only the latest version of each key. Kafka also deletes records based on the specified time-based or size-based log retention settings.

For example, in the following diagram only the latest message (with the highest offset) for a specific message key is retained up to the compaction point. If there are any records remaining up to the retention point they are deleted. In this case, the compaction process would remove all duplicates.

Figure 3.3. Log retention point and compaction point



3.7. MANAGING EFFICIENT DISK UTILIZATION FOR COMPACTION

When employing the compact policy and log cleaner to handle topic logs in Kafka, consider optimizing memory allocation.

You can fine-tune memory allocation using the deduplication property (**dedupe.buffer.size**), which determines the total memory allocated for cleanup tasks across all log cleaner threads. Additionally, you can establish a maximum memory usage limit by defining a percentage through the **buffer.load.factor** property.

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

Each log entry uses exactly 24 bytes, so you can work out how many log entries the buffer can handle in a single run and adjust the setting accordingly.

If possible, consider increasing the number of log cleaner threads if you are looking to reduce the log cleaning time:

```
# ...
log.cleaner.threads=8
# ...
```

If you are experiencing issues with 100% disk bandwidth usage, you can throttle the log cleaner I/O so that the sum of the read/write operations is less than a specified double value based on the capabilities of the disks performing the operations:

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

3.8. HANDLING LARGE MESSAGE SIZES

The default batch size for messages is 1MB, which is optimal for maximum throughput in most use cases. Kafka can accommodate larger batches at a reduced throughput, assuming adequate disk capacity.

Large message sizes are handled in four ways:

1. Producer-side message compression writes compressed messages to the log.
2. Reference-based messaging sends only a reference to data stored in some other system in the message's value.
3. Inline messaging splits messages into chunks that use the same key, which are then combined on output using a stream-processor like Kafka Streams.
4. Broker and producer/consumer client application configuration built to handle larger message sizes.

The reference-based messaging and message compression options are recommended and cover most situations. With any of these options, care must be take to avoid introducing performance issues.

Producer-side compression

For producer configuration, you specify a **compression.type**, such as Gzip, which is then applied to batches of data generated by the producer. Using the broker configuration **compression.type=producer**, the broker retains whatever compression the producer used. Whenever producer and topic compression do not match, the broker has to compress batches again prior to appending them to the log, which impacts broker performance.

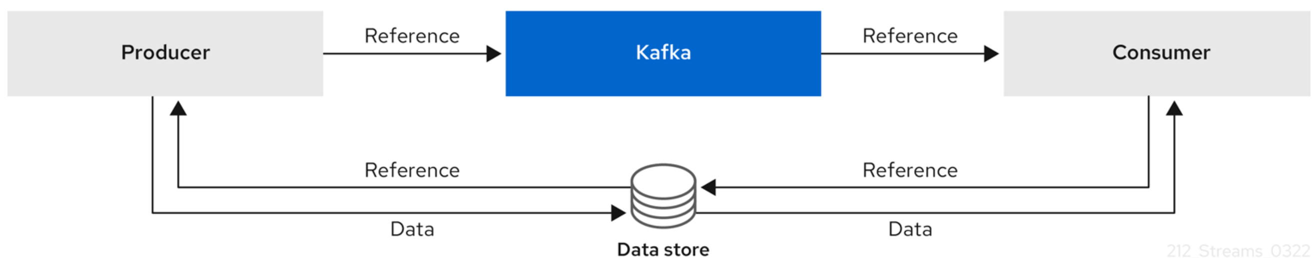
Compression also adds additional processing overhead on the producer and decompression overhead on the consumer, but includes more data in a batch, so is often beneficial to throughput when message data compresses well.

Combine producer-side compression with fine-tuning of the batch size to facilitate optimum throughput. Using metrics helps to gauge the average batch size needed.

Reference-based messaging

Reference-based messaging is useful for data replication when you do not know how big a message will be. The external data store must be fast, durable, and highly available for this configuration to work. Data is written to the data store and a reference to the data is returned. The producer sends a message containing the reference to Kafka. The consumer gets the reference from the message and uses it to fetch the data from the data store.

Figure 3.4. Reference-based messaging flow



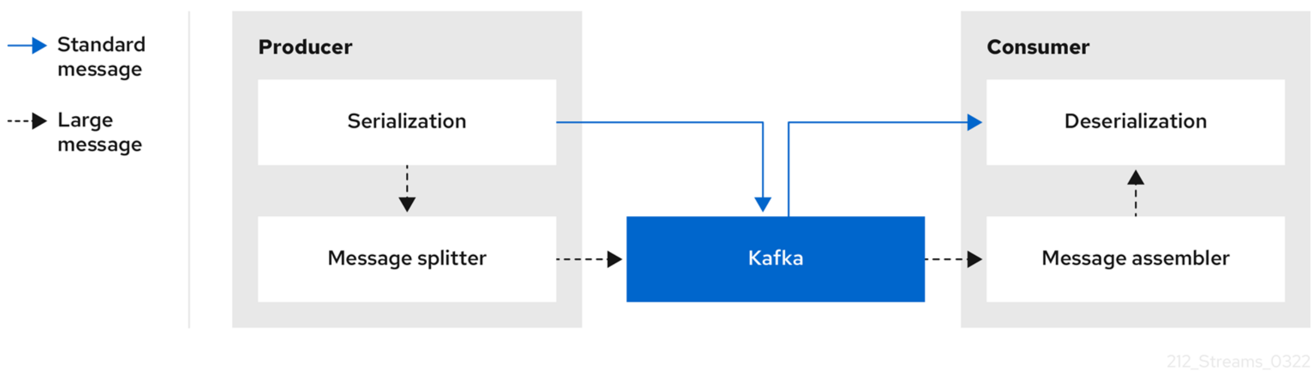
As the message passing requires more trips, end-to-end latency will increase. Another significant drawback of this approach is there is no automatic clean up of the data in the external system when the Kafka message gets cleaned up. A hybrid approach would be to only send large messages to the data store and process standard-sized messages directly.

Inline messaging

Inline messaging is complex, but it does not have the overhead of depending on external systems like reference-based messaging.

The producing client application has to serialize and then chunk the data if the message is too big. The producer then uses the Kafka **ByteArraySerializer** or similar to serialize each chunk again before sending it. The consumer tracks messages and buffers chunks until it has a complete message. The consuming client application receives the chunks, which are assembled before deserialization. Complete messages are delivered to the rest of the consuming application in order according to the offset of the first or last chunk for each set of chunked messages. Successful delivery of the complete message is checked against offset metadata to avoid duplicates during a rebalance.

Figure 3.5. Inline messaging flow



Inline messaging has a performance overhead on the consumer side because of the buffering required, particularly when handling a series of large messages in parallel. The chunks of large messages can become interleaved, so that it is not always possible to commit when all the chunks of a message have been consumed if the chunks of another large message in the buffer are incomplete. For this reason, the buffering is usually supported by persisting message chunks or by implementing commit logic.

Configuration to handle larger messages

If larger messages cannot be avoided, and to avoid blocks at any point of the message flow, you can increase message limits. To do this, configure **message.max.bytes** at the topic level to set the maximum record batch size for individual topics. If you set **message.max.bytes** at the broker level, larger messages are allowed for all topics.

The broker will reject any message that is greater than the limit set with **message.max.bytes**. The buffer size for the producers (**max.request.size**) and consumers (**message.max.bytes**) must be able to accommodate the larger messages.

3.9. CONTROLLING THE LOG FLUSH OF MESSAGE DATA

Generally, the recommendation is to not set explicit flush thresholds and let the operating system perform background flush using its default settings. Partition replication provides greater data durability than writes to any single disk, as a failed broker can recover from its in-sync replicas.

Log flush properties control the periodic writes of cached message data to disk. The scheduler specifies the frequency of checks on the log cache in milliseconds:

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

You can control the frequency of the flush based on the maximum amount of time that a message is kept in-memory and the maximum number of messages in the log before writing to disk:

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

The wait between flushes includes the time to make the check and the specified interval before the flush is carried out. Increasing the frequency of flushes can affect throughput.

If you are using application flush management, setting lower flush thresholds might be appropriate if you are using faster disks.

3.10. PARTITION REBALANCING FOR AVAILABILITY

Partitions can be replicated across brokers for fault tolerance. For a given partition, one broker is elected leader and handles all produce requests (writes to the log). Partition followers on other brokers replicate the partition data of the partition leader for data reliability in the event of the leader failing.

Followers do not normally serve clients, though **rack** configuration allows a consumer to consume messages from the closest replica when a Kafka cluster spans multiple datacenters. Followers operate only to replicate messages from the partition leader and allow recovery should the leader fail. Recovery requires an in-sync follower. Followers stay in sync by sending fetch requests to the leader, which returns messages to the follower in order. The follower is considered to be in sync if it has caught up with the most recently committed message on the leader. The leader checks this by looking at the last offset requested by the follower. An out-of-sync follower is usually not eligible as a leader should the current leader fail, unless unclean leader election is allowed.

You can adjust the lag time before a follower is considered out of sync:

```
# ...
replica.lag.time.max.ms=30000
# ...
```

Lag time puts an upper limit on the time to replicate a message to all in-sync replicas and how long a producer has to wait for an acknowledgment. If a follower fails to make a fetch request and catch up with the latest message within the specified lag time, it is removed from in-sync replicas. You can reduce the lag time to detect failed replicas sooner, but by doing so you might increase the number of followers that fall out of sync needlessly. The right lag time value depends on both network latency and broker disk bandwidth.

When a leader partition is no longer available, one of the in-sync replicas is chosen as the new leader. The first broker in a partition's list of replicas is known as the *preferred* leader. By default, Kafka is enabled for automatic partition leader rebalancing based on a periodic check of leader distribution. That is, Kafka checks to see if the preferred leader is the *current* leader. A rebalance ensures that leaders are evenly distributed across brokers and brokers are not overloaded.

You can use Cruise Control for AMQ Streams to figure out replica assignments to brokers that balance load evenly across the cluster. Its calculation takes into account the differing load experienced by leaders and followers. A failed leader affects the balance of a Kafka cluster because the remaining brokers get the extra work of leading additional partitions.

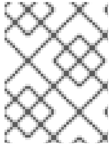
For the assignment found by Cruise Control to actually be balanced it is necessary that partitions are lead by the preferred leader. Kafka can automatically ensure that the preferred leader is being used (where possible), changing the current leader if necessary. This ensures that the cluster remains in the balanced state found by Cruise Control.

You can control the frequency, in seconds, of the rebalance check and the maximum percentage of imbalance allowed for a broker before a rebalance is triggered.

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

The percentage leader imbalance for a broker is the ratio between the current number of partitions for which the broker is the current leader and the number of partitions for which it is the preferred leader. You can set the percentage to zero to ensure that preferred leaders are always elected, assuming they are in sync.

If the checks for rebalances need more control, you can disable automated rebalances. You can then choose when to trigger a rebalance using the **kafka-leader-election.sh** command line tool.



NOTE

The Grafana dashboards provided with AMQ Streams show metrics for under-replicated partitions and partitions that do not have an active leader.

3.11. UNCLEAN LEADER ELECTION

Leader election to an in-sync replica is considered clean because it guarantees no loss of data. And this is what happens by default. But what if there is no in-sync replica to take on leadership? Perhaps the ISR (in-sync replica) only contained the leader when the leader's disk died. If a minimum number of in-sync replicas is not set, and there are no followers in sync with the partition leader when its hard drive fails irrevocably, data is already lost. Not only that, but *a new leader cannot be elected* because there are no in-sync followers.

You can configure how Kafka handles leader failure:

```
# ...
unclean.leader.election.enable=false
# ...
```

Unclean leader election is disabled by default, which means that out-of-sync replicas cannot become leaders. With clean leader election, if no other broker was in the ISR when the old leader was lost, Kafka waits until that leader is back online before messages can be written or read. Unclean leader election means out-of-sync replicas can become leaders, but you risk losing messages. The choice you make depends on whether your requirements favor availability or durability.

You can override the default configuration for specific topics at the topic level. If you cannot afford the risk of data loss, then leave the default configuration.

3.12. AVOIDING UNNECESSARY CONSUMER GROUP REBALANCES

For consumers joining a new consumer group, you can add a delay so that unnecessary rebalances to the broker are avoided:

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

The delay is the amount of time that the coordinator waits for members to join. The longer the delay, the more likely it is that all the members will join in time and avoid a rebalance. But the delay also prevents the group from consuming until the period has ended.

CHAPTER 4. KAFKA CONSUMER CONFIGURATION TUNING

Use a basic consumer configuration with optional properties that are tailored to specific use cases.

When tuning your consumers your primary concern will be ensuring that they cope efficiently with the amount of data ingested. As with the producer tuning, be prepared to make incremental changes until the consumers operate as expected.

4.1. BASIC CONSUMER CONFIGURATION

Connection and deserializer properties are required for every consumer. Generally, it is good practice to add a client id for tracking.

In a consumer configuration, irrespective of any subsequent configuration:

- The consumer fetches from a given offset and consumes the messages in order, unless the offset is changed to skip or re-read messages.
- The broker does not know if the consumer processed the responses, even when committing offsets to Kafka, because the offsets might be sent to a different broker in the cluster.

Basic consumer configuration properties

```
# ...
bootstrap.servers=localhost:9092 (1)
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer (2)
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer (3)
client.id=my-client (4)
group.id=my-group-id (5)
# ...
```

1. (Required) Tells the consumer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The consumer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it is not necessary to provide a list of all the brokers in the cluster. If you are using a loadbalancer service to expose the Kafka cluster, you only need the address for the service because the availability is handled by the loadbalancer.
2. (Required) Deserializer to transform the bytes fetched from the Kafka broker into message keys.
3. (Required) Deserializer to transform the bytes fetched from the Kafka broker into message values.
4. (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request. The id can also be used to throttle consumers based on processing time quotas.
5. (Conditional) A group id is *required* for a consumer to be able to join a consumer group.

4.2. SCALING DATA CONSUMPTION USING CONSUMER GROUPS

Consumer groups share a typically large data stream generated by one or multiple producers from a given topic. Consumers are grouped using a **group.id** property, allowing messages to be spread across the members. One of the consumers in the group is elected leader and decides how the partitions are assigned to the consumers in the group. Each partition can only be assigned to a single consumer.

If you do not already have as many consumers as partitions, you can scale data consumption by adding more consumer instances with the same **group.id**. Adding more consumers to a group than there are partitions will not help throughput, but it does mean that there are consumers on standby should one stop functioning. If you can meet throughput goals with fewer consumers, you save on resources.

Consumers within the same consumer group send offset commits and heartbeats to the same broker. So the greater the number of consumers in the group, the higher the request load on the broker.

```
# ...
group.id=my-group-id 1
# ...
```

1. Add a consumer to a consumer group using a group id.

4.3. MESSAGE ORDERING GUARANTEES

Kafka brokers receive fetch requests from consumers that ask the broker to send messages from a list of topics, partitions and offset positions.

A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka **only** provides ordering guarantees for messages in a single partition. Conversely, if a consumer is consuming messages from multiple partitions, the order of messages in different partitions as observed by the consumer does not necessarily reflect the order in which they were sent.

If you want a strict ordering of messages from one topic, use one partition per consumer.

4.4. OPTIMIZING CONSUMERS FOR THROUGHPUT AND LATENCY

Control the number of messages returned when your client application calls **KafkaConsumer.poll()**.

Use the **fetch.max.wait.ms** and **fetch.min.bytes** properties to increase the minimum amount of data fetched by the consumer from the Kafka broker. Time-based batching is configured using **fetch.max.wait.ms**, and size-based batching is configured using **fetch.min.bytes**.

If CPU utilization in the consumer or broker is high, it might be because there are too many requests from the consumer. You can adjust **fetch.max.wait.ms** and **fetch.min.bytes** properties higher so that there are fewer requests and messages are delivered in bigger batches. By adjusting higher, throughput is improved with some cost to latency. You can also adjust higher if the amount of data being produced is low.

For example, if you set **fetch.max.wait.ms** to 500ms and **fetch.min.bytes** to 16384 bytes, when Kafka receives a fetch request from the consumer it will respond when the first of either threshold is reached.

Conversely, you can adjust the **fetch.max.wait.ms** and **fetch.min.bytes** properties lower to improve end-to-end latency.

```
# ...
fetch.max.wait.ms=500 1
fetch.min.bytes=16384 2
# ...
```

1. The maximum time in milliseconds the broker will wait before completing fetch requests. The default is **500** milliseconds.
2. If a minimum batch size in bytes is used, a request is sent when the minimum is reached, or messages have been queued for longer than **fetch.max.wait.ms** (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

Lowering latency by increasing the fetch request size

Use the **fetch.max.bytes** and **max.partition.fetch.bytes** properties to increase the maximum amount of data fetched by the consumer from the Kafka broker.

The **fetch.max.bytes** property sets a maximum limit in bytes on the amount of data fetched from the broker at one time.

The **max.partition.fetch.bytes** sets a maximum limit in bytes on how much data is returned for each partition, which must always be larger than the number of bytes set in the broker or topic configuration for **max.message.bytes**.

The maximum amount of memory a client can consume is calculated approximately as:

> *NUMBER-OF-BROKERS* * fetch.max.bytes and *NUMBER-OF-PARTITIONS* * max.partition.fetch.bytes

If memory usage can accommodate it, you can increase the values of these two properties. By allowing more data in each request, latency is improved as there are fewer fetch requests.

```
# ...
fetch.max.bytes=52428800 (1)
max.partition.fetch.bytes=1048576 (2)
# ...
```

1. The maximum amount of data in bytes returned for a fetch request.
2. The maximum amount of data in bytes returned for each partition.

4.5. AVOIDING DATA LOSS OR DUPLICATION WHEN COMMITTING OFFSETS

The Kafka *auto-commit mechanism* allows a consumer to commit the offsets of messages automatically. If enabled, the consumer will commit offsets received from polling the broker at 5000ms intervals.

The auto-commit mechanism is convenient, but it introduces a risk of data loss and duplication. If a consumer has fetched and transformed a number of messages, but the system crashes with processed messages in the consumer buffer when performing an auto-commit, that data is lost. If the system crashes after processing the messages, but before performing the auto-commit, the data is duplicated on another consumer instance after rebalancing.

Auto-committing can avoid data loss only when all messages are processed before the next poll to the broker, or the consumer closes.

To minimize the likelihood of data loss or duplication, you can set **enable.auto.commit** to **false** and develop your client application to have more control over committing offsets. Or you can use **auto.commit.interval.ms** to decrease the intervals between commits.

```
# ...
enable.auto.commit=false (1)
# ...
```

(1) Auto commit is set to false to provide more control over committing offsets.

By setting to **enable.auto.commit** to **false**, you can commit offsets after **all** processing has been performed and the message has been consumed. For example, you can set up your application to call the Kafka **commitSync** and **commitAsync** commit APIs.

The **commitSync** API commits the offsets in a message batch returned from polling. You call the API when you are finished processing all the messages in the batch. If you use the **commitSync** API, the application will not poll for new messages until the last offset in the batch is committed. If this negatively affects throughput, you can commit less frequently, or you can use the **commitAsync** API. The **commitAsync** API does not wait for the broker to respond to a commit request, but risks creating more duplicates when rebalancing. A common approach is to combine both commit APIs in an application, with the **commitSync** API used just before shutting the consumer down or rebalancing to make sure the final commit is successful.

4.5.1. Controlling transactional messages

Consider using transactional ids and enabling idempotence (**enable.idempotence=true**) on the producer side to guarantee exactly-once delivery. On the consumer side, you can then use the **isolation.level** property to control how transactional messages are read by the consumer.

The **isolation.level** property has two valid values:

- **read_committed**
- **read_uncommitted** (default)

Use **read_committed** to ensure that only transactional messages that have been committed are read by the consumer. However, this will cause an increase in end-to-end latency, because the consumer will not be able to return a message until the brokers have written the transaction markers that record the result of the transaction (*committed* or *aborted*).

```
# ...
enable.auto.commit=false
isolation.level=read_committed (1)
# ...
```

(1) Set to **read_committed** so that only committed messages are read by the consumer.

4.6. RECOVERING FROM FAILURE TO AVOID DATA LOSS

Use the **session.timeout.ms** and **heartbeat.interval.ms** properties to configure the time taken to check and recover from consumer failure within a consumer group.

The **session.timeout.ms** property specifies the maximum amount of time in milliseconds a consumer

within a consumer group can be out of contact with a broker before being considered inactive and a *rebalancing* is triggered between the active consumers in the group. When the group rebalances, the partitions are reassigned to the members of the group.

The **heartbeat.interval.ms** property specifies the interval in milliseconds between *heartbeat* checks to the consumer group coordinator to indicate that the consumer is active and connected. The heartbeat interval must be lower, usually by a third, than the session timeout interval.

If you set the **session.timeout.ms** property lower, failing consumers are detected earlier, and rebalancing can take place quicker. However, take care not to set the timeout so low that the broker fails to receive a heartbeat in time and triggers an unnecessary rebalance.

Decreasing the heartbeat interval reduces the chance of accidental rebalancing, but more frequent heartbeats increases the overhead on broker resources.

4.7. MANAGING OFFSET POLICY

Use the **auto.offset.reset** property to control how a consumer behaves when no offsets have been committed, or a committed offset is no longer valid or deleted.

Suppose you deploy a consumer application for the first time, and it reads messages from an existing topic. Because this is the first time the **group.id** is used, the **__consumer_offsets** topic does not contain any offset information for this application. The new application can start processing all existing messages from the start of the log or only new messages. The default reset value is **latest**, which starts at the end of the partition, and consequently means some messages are missed. To avoid data loss, but increase the amount of processing, set **auto.offset.reset** to **earliest** to start at the beginning of the partition.

Also consider using the **earliest** option to avoid messages being lost when the offsets retention period (**offsets.retention.minutes**) configured for a broker has ended. If a consumer group or standalone consumer is inactive and commits no offsets during the retention period, previously committed offsets are deleted from **__consumer_offsets**.

```
# ...
heartbeat.interval.ms=3000 (1)
session.timeout.ms=45000 (2)
auto.offset.reset=earliest (3)
# ...
```

1. Adjust the heartbeat interval lower according to anticipated rebalances.

2. If no heartbeats are received by the Kafka broker before the timeout duration expires, the consumer is removed from the consumer group and a rebalance is initiated. If the broker configuration has a **group.min.session.timeout.ms** and **group.max.session.timeout.ms**, the session timeout value must be within that range.

3. Set to **earliest** to return to the start of a partition and avoid data loss if offsets were not committed.

If the amount of data returned in a single fetch request is large, a timeout might occur before the consumer has processed it. In this case, you can lower **max.partition.fetch.bytes** or increase **session.timeout.ms**.

4.8. MINIMIZING THE IMPACT OF REBALANCES

The rebalancing of a partition between active consumers in a group is the time it takes for:

- Consumers to commit their offsets
- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

Clearly, the process increases the downtime of a service, particularly when it happens repeatedly during a rolling restart of a consumer group cluster.

In this situation, you can use the concept of *static membership* to reduce the number of rebalances. Rebalancing assigns topic partitions evenly among consumer group members. Static membership uses persistence so that a consumer instance is recognized during a restart after a session timeout.

The consumer group coordinator can identify a new consumer instance using a unique id that is specified using the **group.instance.id** property. During a restart, the consumer is assigned a new member id, but as a static member it continues with the same instance id, and the same assignment of topic partitions is made.

If the consumer application does not make a call to poll at least every **max.poll.interval.ms** milliseconds, the consumer is considered to be failed, causing a rebalance. If the application cannot process all the records returned from poll in time, you can avoid a rebalance by using the **max.poll.interval.ms** property to specify the interval in milliseconds between polls for new messages from a consumer. Or you can use the **max.poll.records** property to set a maximum limit on the number of records returned from the consumer buffer, allowing your application to process fewer records within the **max.poll.interval.ms** limit.

```
# ...
group.instance.id=UNIQUE-ID (1)
max.poll.interval.ms=300000 (2)
max.poll.records=500 (3)
# ...
```

1. The unique instance id ensures that a new consumer instance receives the same assignment of topic partitions.
2. Set the interval to check the consumer is continuing to process messages.
3. Sets the number of processed records returned from the consumer.

CHAPTER 5. KAFKA PRODUCER CONFIGURATION TUNING

Use a basic producer configuration with optional properties that are tailored to specific use cases.

Adjusting your configuration to maximize throughput might increase latency or vice versa. You will need to experiment and tune your producer configuration to get the balance you need.

5.1. BASIC PRODUCER CONFIGURATION

Connection and serializer properties are required for every producer. Generally, it is good practice to add a client id for tracking, and use compression on the producer to reduce batch sizes in requests.

In a basic producer configuration:

- The order of messages in a partition is not guaranteed.
- The acknowledgment of messages reaching the broker does not guarantee durability.

Basic producer configuration properties

```
# ...
bootstrap.servers=localhost:9092 1
key.serializer=org.apache.kafka.common.serialization.StringSerializer 2
value.serializer=org.apache.kafka.common.serialization.StringSerializer 3
client.id=my-client 4
compression.type=gzip 5
# ...
```

1. (Required) Tells the producer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The producer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it's not necessary to provide a list of all the brokers in the cluster.
2. (Required) Serializer to transform the key of each message to bytes prior to them being sent to a broker.
3. (Required) Serializer to transform the value of each message to bytes prior to them being sent to a broker.
4. (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request.
5. (Optional) The codec for compressing messages, which are sent and might be stored in compressed format and then decompressed when reaching a consumer. Compression is useful for improving throughput and reducing the load on storage, but might not be suitable for low latency applications where the cost of compression or decompression could be prohibitive.

5.2. DATA DURABILITY

Message delivery acknowledgments minimize the likelihood that messages are lost. Acknowledgments are enabled by default with the **acks** property set at **acks=all**.

Acknowledging message delivery

```
# ...
acks=all (1)
# ...
```

(1) **acks=all** forces a leader replica to replicate messages to a certain number of followers before acknowledging that the message request was successfully received.

The **acks=all** setting offers the strongest guarantee of delivery, but it will increase the latency between the producer sending a message and receiving acknowledgment. If you don't require such strong guarantees, a setting of **acks=0** or **acks=1** provides either no delivery guarantees or only acknowledgment that the leader replica has written the record to its log.

With **acks=all**, the leader waits for all in-sync replicas to acknowledge message delivery. A topic's **min.insync.replicas** configuration sets the minimum required number of in-sync replica acknowledgements. The number of acknowledgements include that of the leader and followers.

A typical starting point is to use the following configuration:

- Producer configuration:
    - **acks=all** (default)
- Broker configuration for topic replication:
    - **default.replication.factor=3** (default = **1**)
    - **min.insync.replicas=2** (default = **1**)

When you create a topic, you can override the default replication factor. You can also override **min.insync.replicas** at the topic level in the topic configuration.

AMQ Streams uses this configuration in the example configuration files for multi-node deployment of Kafka.

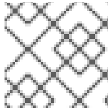
The following table describes how this configuration operates depending on the availability of followers that replicate the leader replica.

Table 5.1. Follower availability

Number of followers available and in-sync	Acknowledgements	Producer can send messages?
2	The leader waits for 2 follower acknowledgements	Yes
1	The leader waits for 1 follower acknowledgement	Yes
0	The leader raises an exception	No

A topic replication factor of 3 creates one leader replica and two followers. In this configuration, the producer can continue if a single follower is unavailable. Some delay can occur whilst removing a failed broker from the in-sync replicas or a creating a new leader. If the second follower is also unavailable,

message delivery will not be successful. Instead of acknowledging successful message delivery, the leader sends an error (*not enough replicas*) to the producer. The producer raises an equivalent exception. With **retries** configuration, the producer can resend the failed message request.



NOTE

If the system fails, there is a risk of unsent data in the buffer being lost.

5.3. ORDERED DELIVERY

Idempotent producers avoid duplicates as messages are delivered exactly once. IDs and sequence numbers are assigned to messages to ensure the order of delivery, even in the event of failure. If you are using **acks=all** for data consistency, using idempotency makes sense for ordered delivery. Idempotency is enabled for producers by default. With idempotency enabled, you can set the number of concurrent in-flight requests to a maximum of 5 for message ordering to be preserved.

Ordered delivery with idempotency

```
# ...
enable.idempotence=true (1)
max.in.flight.requests.per.connection=5 (2)
acks=all (3)
retries=2147483647 (4)
# ...
```

1. Set to **true** to enable the idempotent producer.
2. With idempotent delivery the number of in-flight requests may be greater than 1 while still providing the message ordering guarantee. The default is 5 in-flight requests.
3. Set **acks** to **all**.
4. Set the number of attempts to resend a failed message request.

If you choose not to use **acks=all** and disable idempotency because of the performance cost, set the number of in-flight (unacknowledged) requests to 1 to preserve ordering. Otherwise, a situation is possible where *Message-A* fails only to succeed after *Message-B* was already written to the broker.

Ordered delivery without idempotency

```
# ...
enable.idempotence=false (1)
max.in.flight.requests.per.connection=1 (2)
retries=2147483647
# ...
```

1. Set to **false** to disable the idempotent producer.
2. Set the number of in-flight requests to exactly **1**.

5.4. RELIABILITY GUARANTEES

Idempotence is useful for exactly once writes to a single partition. Transactions, when used with idempotence, allow exactly once writes across multiple partitions.

Transactions guarantee that messages using the same transactional ID are produced once, and either *all* are successfully written to the respective logs or *none* of them are.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

1. Specify a unique transactional ID.
2. Set the maximum allowed time for transactions in milliseconds before a timeout error is returned. The default is **900000** or 15 minutes.

The choice of **transactional.id** is important in order that the transactional guarantee is maintained. Each transactional id should be used for a unique set of topic partitions. For example, this can be achieved using an external mapping of topic partition names to transactional ids, or by computing the transactional id from the topic partition names using a function that avoids collisions.

5.5. OPTIMIZING PRODUCERS FOR THROUGHPUT AND LATENCY

Usually, the requirement of a system is to satisfy a particular throughput target for a proportion of messages within a given latency. For example, targeting 500,000 messages per second with 95% of messages being acknowledged within 2 seconds.

It’s likely that the messaging semantics (message ordering and durability) of your producer are defined by the requirements for your application. For instance, it’s possible that you don’t have the option of using **acks=0** or **acks=1** without breaking some important property or guarantee provided by your application.

Broker restarts have a significant impact on high percentile statistics. For example, over a long period the 99th percentile latency is dominated by behavior around broker restarts. This is worth considering when designing benchmarks or comparing performance numbers from benchmarking with performance numbers seen in production.

Depending on your objective, Kafka offers a number of configuration parameters and techniques for tuning producer performance for throughput and latency.

Message batching (linger.ms and batch.size)

Message batching delays sending messages in the hope that more messages destined for the same broker will be sent, allowing them to be batched into a single produce request. Batching is a compromise between higher latency in return for higher throughput. Time-based batching is configured using **linger.ms**, and size-based batching is configured using **batch.size**.

Compression (compression.type)

Message compression adds latency in the producer (CPU time spent compressing the messages), but makes requests (and potentially disk writes) smaller, which can increase throughput. Whether compression is worthwhile, and the best compression to use, will depend on the messages being sent.

Compression happens on the thread which calls **KafkaProducer.send()**, so if the latency of this method matters for your application you should consider using more threads.

Pipelining (max.in.flight.requests.per.connection)

Pipelining means sending more requests before the response to a previous request has been received. In general more pipelining means better throughput, up to a threshold at which other effects, such as worse batching, start to counteract the effect on throughput.

Lowering latency

When your application calls **KafkaProducer.send()** the messages are:

- Processed by any interceptors
- Serialized
- Assigned to a partition
- Compressed
- Added to a batch of messages in a per-partition queue

At which point the **send()** method returns. So the time **send()** is blocked is determined by:

- The time spent in the interceptors, serializers and partitioner
- The compression algorithm used
- The time spent waiting for a buffer to use for compression

Batches will remain in the queue until one of the following occurs:

- The batch is full (according to **batch.size**)
- The delay introduced by **linger.ms** has passed
- The sender is about to send message batches for other partitions to the same broker, and it is possible to add this batch too
- The producer is being flushed or closed

Look at the configuration for batching and buffering to mitigate the impact of **send()** blocking on latency.

```
# ...
linger.ms=100 (1)
batch.size=16384 (2)
buffer.memory=33554432 (3)
# ...
```

(1) The **linger** property adds a delay in milliseconds so that larger batches of messages are accumulated and sent in a request. The default is **0'.**

(2) If a maximum **batch.size** in bytes is used, a request is sent when the maximum is reached, or messages have been queued for longer than **linger.ms** (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

3. The buffer size must be at least as big as the batch size, and be able to accommodate buffering, compression and in-flight requests.

Increasing throughput

Improve throughput of your message requests by adjusting the maximum time to wait before a message is delivered and completes a send request.

You can also direct messages to a specified partition by writing a custom partitioner to replace the default.

```
# ...
delivery.timeout.ms=120000 (1)
partitioner.class=my-custom-partitioner (2)

# ...
```

1. The maximum time in milliseconds to wait for a complete send request. You can set the value to **MAX_LONG** to delegate to Kafka an indefinite number of retries. The default is **120000** or 2 minutes.
2. Specify the class name of the custom partitioner.

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ Streams is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing Your Account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a Subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading Zip and Tar Files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **AMQ Streams for Apache Kafka** entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

Revised on 2023-12-06 17:40:09 UTC