Why Is Kafka So Fast

Avoid Random Disk Access

Kafka writes everything onto the disk in order and consumers fetch data in order too. So disk access always works sequentially instead of randomly.

For traditional hard disks(HDD), sequential access is much faster than random access.

Here is a comparison:

hardware	sequential writes	random writes
6 * 7200rpm SATA RAID-5	300MB/s	50KB/s

Kafka Writes Everything Onto The Disk Instead of Memory

Kafka writes everything onto the disk instead of memory.

Isn't memory supposed to be faster than disks? Typically it's the case, for Random Disk Access. But for sequential access, the difference is much smaller.

But still, sequential memory access is faster than Sequential Disk Access, why not choose memory?

Because Kafka runs on top of JVM, which gives us two disadvantages.

- The memory overhead of objects is very high, often doubling the size of the data stored(or even higher).
- Garbage Collection happens every now and then, so creating objects in memory is very expensive as in-heap data increases because we will need more time to collect unused data(which is garbage).
- So writing to file systems may be better than writing to memory. Even better, we can utilize MMAP(memory mapped files) to make it faster.

Memory Mapped Files(MMAP)

Basically, MMAP(Memory Mapped Files) can map the file contents from the disk into memory. And when we write something into the mapped memory, the OS will flush the change onto the disk sometime later.

So everything is faster because we are using memory actually, but in an indirect way.

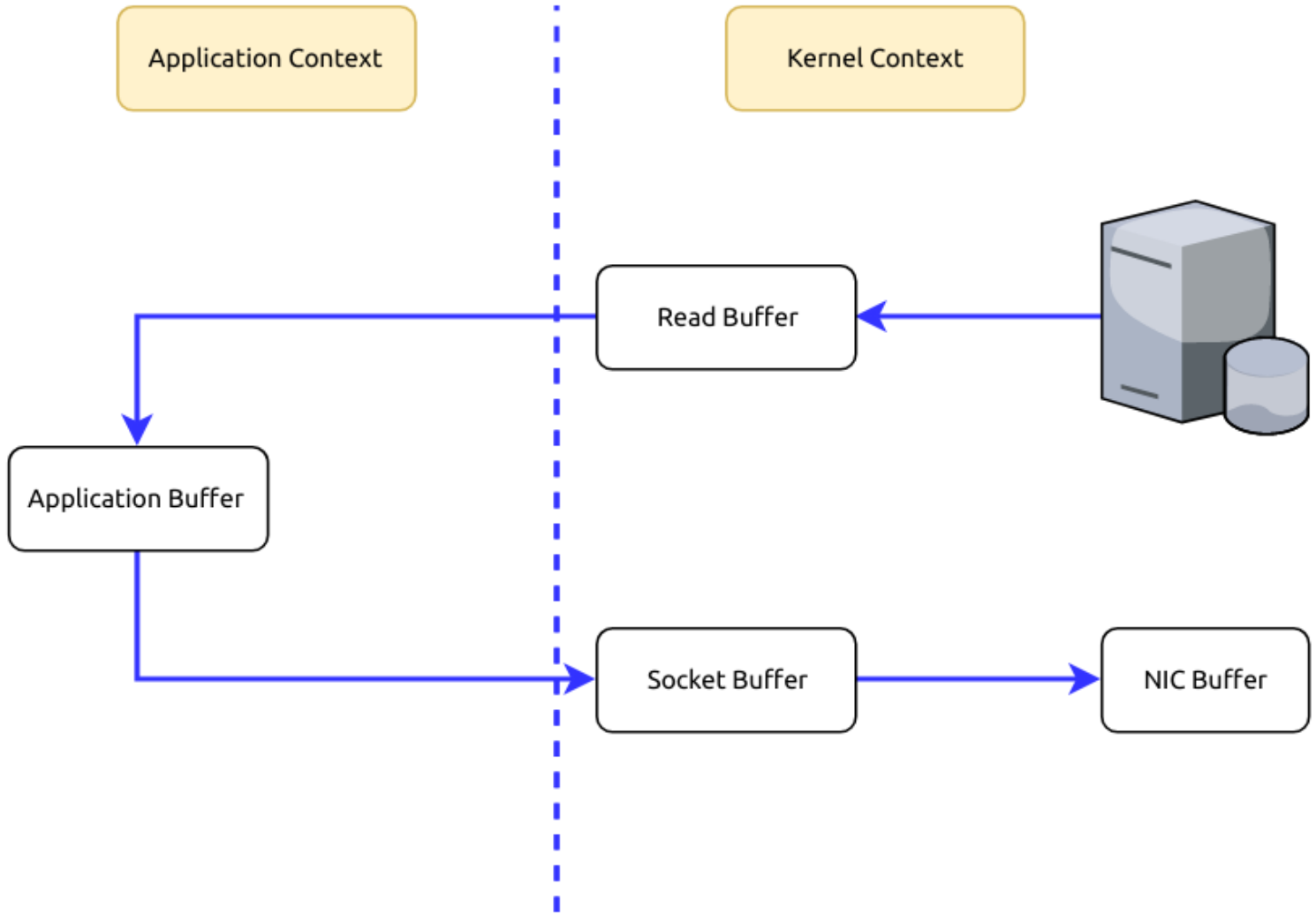
No Zero Copy

Suppose that we are fetching data from the memory and sending them to the Internet.

What is happening in the process is usually twofold.

To fetch data from the memory, we need to copy those data from the Kernel Context into the Application Context.

To send those data to the Internet, we need to copy the data from the Application Context into the Kernel Context.

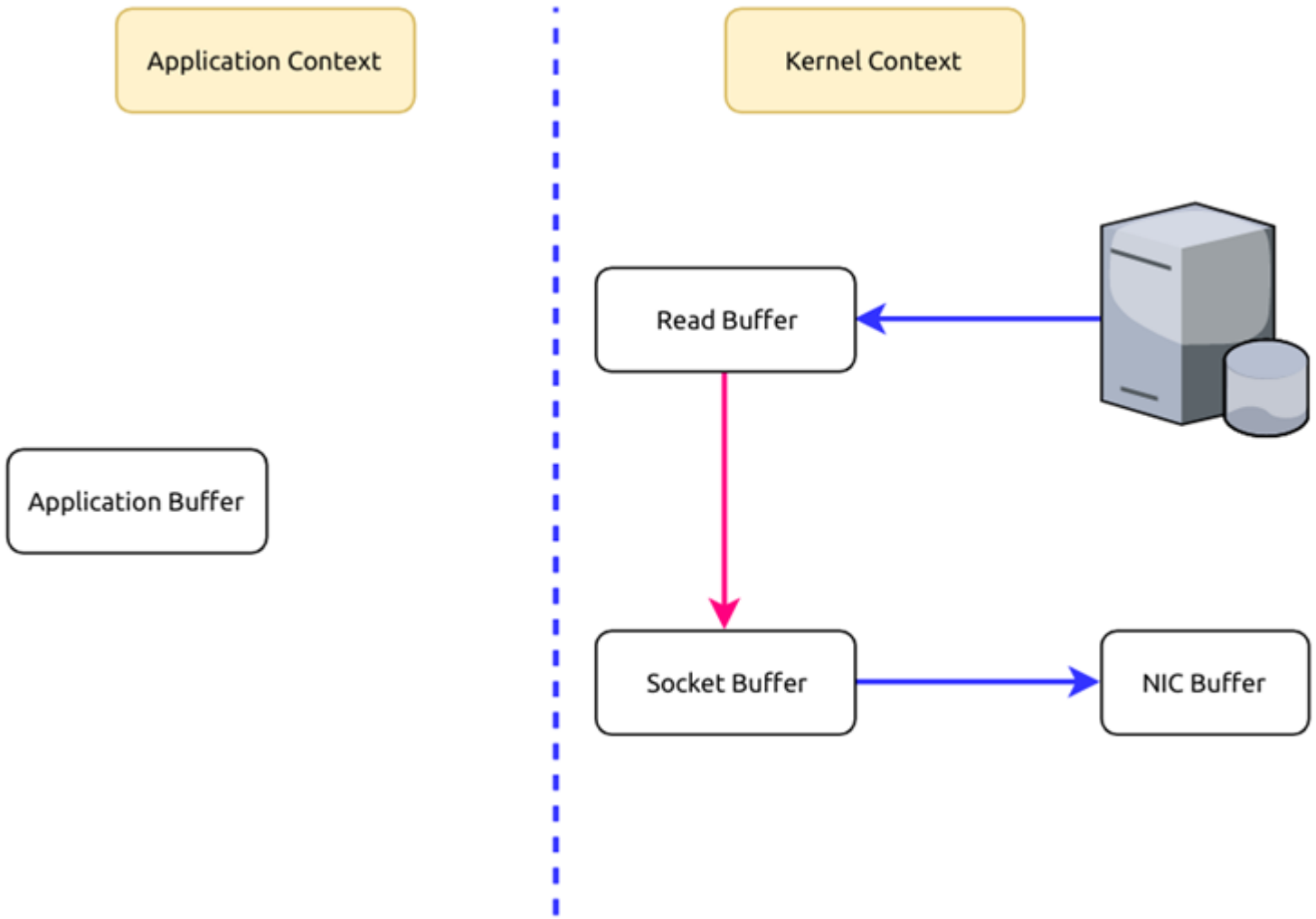

Application Context
Kernel Context
Read Buffer
Application Buffer
Socket Buffer
NIC Buffer
no zero copy

Zero Copy

As you can see, it's redundant to copy data between the Kernel Context and the Application Context.

Can we avoid it? Yes,

using Zero Copy we can copy data directly from the Kernel Context to the Kernel Context.


Application Context
Kernel Context
Read Buffer
Application Buffer
Socket Buffer
NIC Buffer

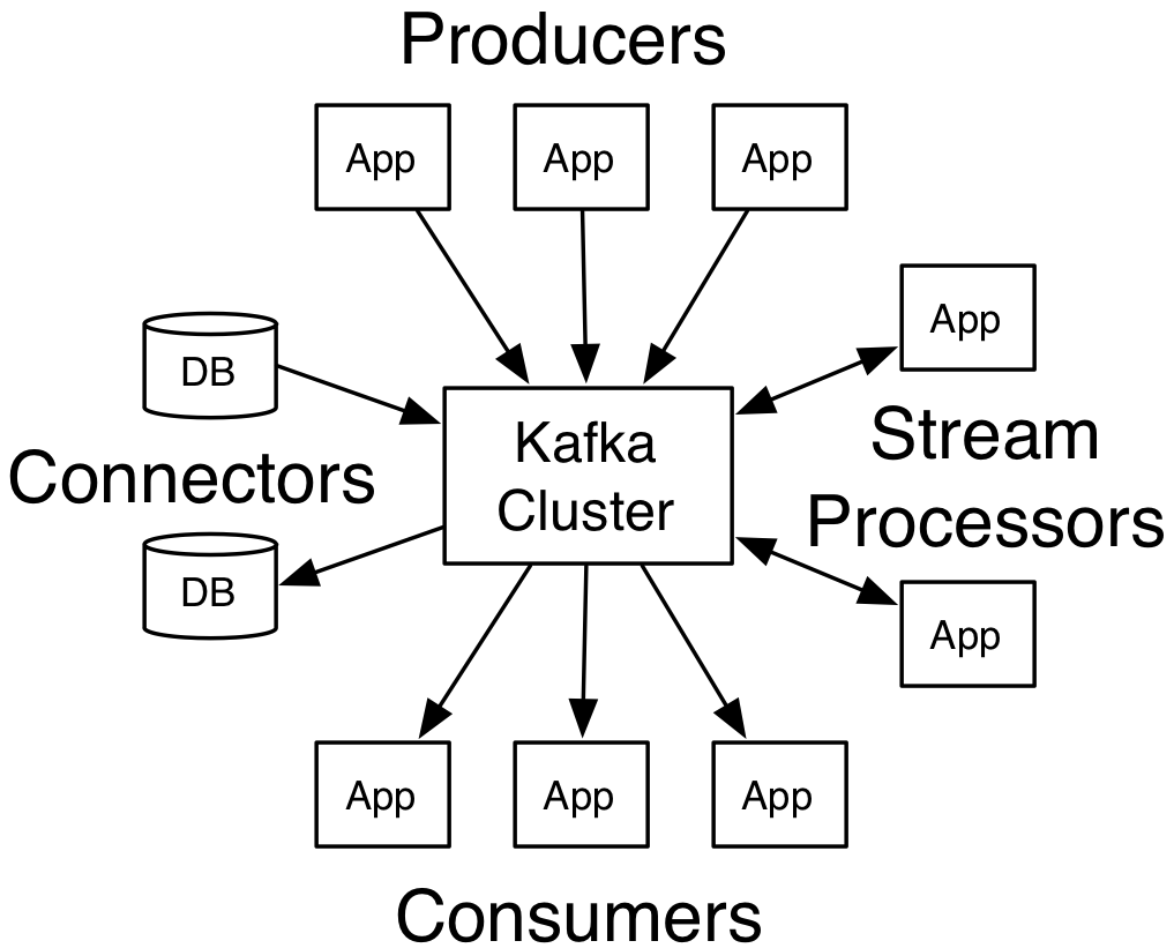
zero copy

Batch Data

Kafka only sends data when batch.size is reached instead of one by one. Assuming the bandwidth is 10MB/s, sending 10MB data in one go is much faster than sending 10000 messages one by one(assuming each message takes 100 bytes).

Kafka core APIs:

Kafka has four core APIs:



- The Producer API allows an application to publish a stream of records to one or more Kafka topics.

- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.

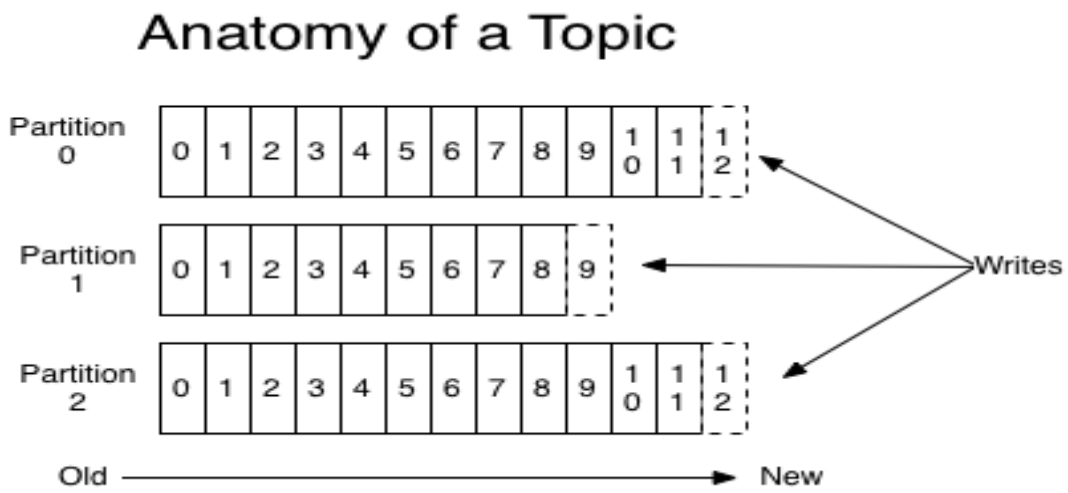
- The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

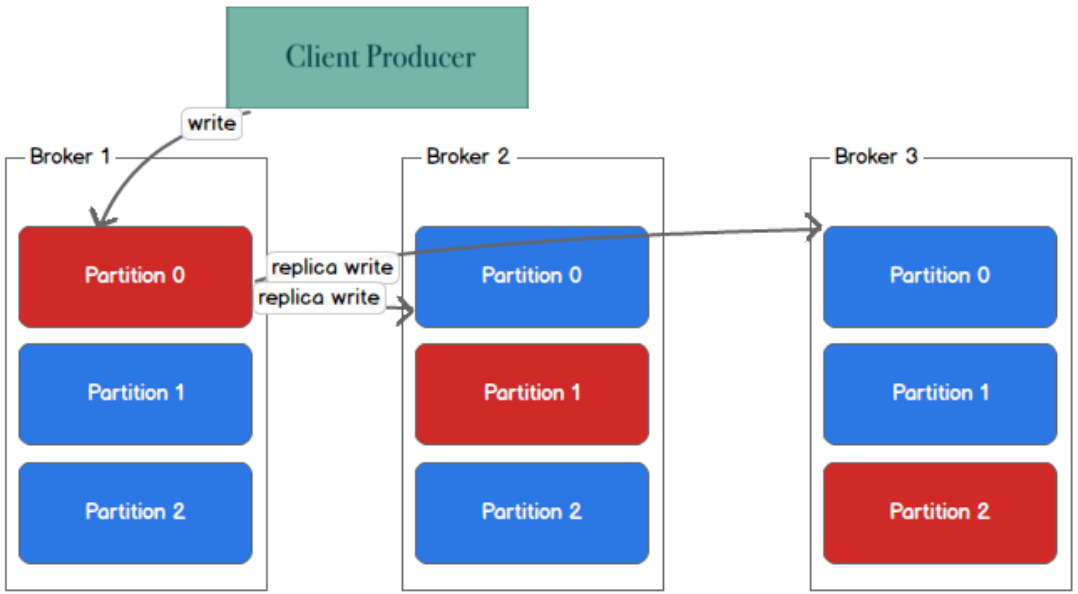
  For example, a connector to a relational database might capture every change to a table.

Topics and Logs

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Kafka Topic Architecture - Replication, Failover, and Parallel Processing



Leader (red)
replicas (blue)

Record is considered "committed" when all ISRs for partition wrote to their log.

Only committed records are readable from consumer

There are two types of replicas:

Leader replica

Each partition has a single replica designated as the leader. All produce and consume requests go through the leader, in order to guarantee consistency.

The leader handles all read and write requests for the partition

Follower replica

All replicas for a partition that are not leaders are called followers.

Followers don't serve client requests; their only job is to replicate messages from the leader and stay up-to-date with the most recent messages the leader has.

In the event that a leader replica for a partition crashes, one of the follower replicas will be promoted to become the new leader for the partition.

Each server acts as a leader for some of its partitions and a follower for others, so load is well balanced within the cluster

Replicas that are consistently asking for the latest messages, is called in-sync replicas.

Only in-sync replicas are eligible to be elected as partition leaders in case the existing leader fails.

The amount of time a follower can be inactive or behind before it is considered out of sync is controlled by the replica.lag.time.max.ms configuration parameter. Default Value is 500.

In addition to the current leader, each partition has a preferred leader—the replica that was the leader when the topic was originally created. It is preferred because when partitions are first created, the leaders are balanced between brokers

By default, Kafka is configured with auto.leader.rebalance.enable=true, which will check if the preferred leader replica is not the current leader but is in-sync and trigger leader election to make the preferred leader the current leader.

kafka-preferred-replica-election

Note : However this option is not recommended.
Manually call the preferred replica election script like this

/bin/kafka-preferred-replica-election.sh --zookeeper (for all topics) or use the command with --path-to-json-file move.json

Finding the Preferred Leaders The best way to identify the current preferred leader is by looking at the list of replicas for a partition and replicas in the output of the **kafka-topics.sh tool**.

Graceful shutdown

Graceful shutdown

The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine. This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes.

When a server is stopped gracefully it has two optimizations it will take advantage of:

It will sync all its logs to disk to avoid needing to do any log recovery when it restarts (i.e. validating the checksum for all messages in the tail of the log). Log recovery takes time so this speeds up intentional restarts.

It will migrate any partitions the server is the leader for to other replicas prior to shutting down. This will make the leadership transfer faster and minimize the time each partition is unavailable to a few milliseconds.

Syncing the logs will happen automatically whenever the server is stopped other than by a hard kill, but the controlled leadership migration requires using a special setting:

controlled.shutdown.enable=true

Note that controlled shutdown will only succeed if all the partitions hosted on the broker have replicas (i.e. the replication factor is greater than 1 and at least one of these replicas is alive).

- Run kafka-server-stop.sh
- (OR) Use kill command with -s to signal gracefull shutdown of the process.

Partitions, Segments & Indexes

Partitions and Segments

Topics are made of Partitions

Partitions are made of segments (files)

Ex :

	Partition-0	
segment 0:	segment: 1	segment: 2
Offset 0-200	Offset 201-400	Offset 401-? (Active)

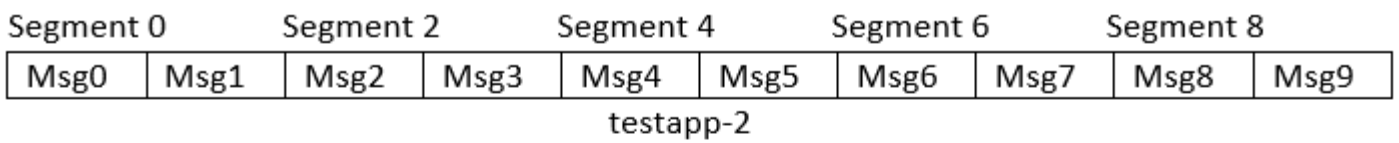
Note : Only one segment is ACTIVE per partition

Segment and Indexes

Segments come with two indexes (files)

- An offset to position index: allows Kafka where to read to find a message
- A timestamp to offset index: allows Kafka to find messages with a timestamp

This will help Kafka to find the data in a constant time.



Index File

Offset	Position
0	0
1	200

Log File

Offset	Position	Time	Message
0	0	1222333333333	Faith
1	200	1222233333444	Trust

We need not to worry about preserving logs then we can set cleanup.policy to ‘delete’ and let Kafka remove log files after a set time or after they reach a certain size.

If we do need to keep logs around for a while, use the ‘compact’ policy

Log Cleanup Policies

Kafka data will get expired, based on the policies:

Policy 1 : log.cleanup.policy=delete (default for all topics)

- ❑ Delete based on age of data (Default is a week)
- ❑ Delete based on max size of log (default -1 i.e. infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic_Consumer_offsets)

- ❑ Delete based on keys of your messages
- ❑ will delete old duplicate keys after that active segment is committed

log.segment.bytes : the max size of a single segment in bytes (default: 1GB)

A smaller log.segment.bytes means :

- ❑ More segments per partitions
- ❑ Log compaction happens more often
- ❑ But Kafka has to keep more files opened (Error: Too many open files)

(decision is made on how fast will i have new segments based on throughput?)

log.segment.ms: the time kafka will wait before committing if the segment is not full (default: 1 week)

A smaller log.segment.ms means:

- ❑ You set a max frequency for log compaction (more frequent triggers)
- ❑ Maybe you want daily compaction instead of weekly?

(Decision is made on how often do i need log compaction to happen?)

log.retention.hours
number of hours to keep data for (default is 168 hrs / one week)

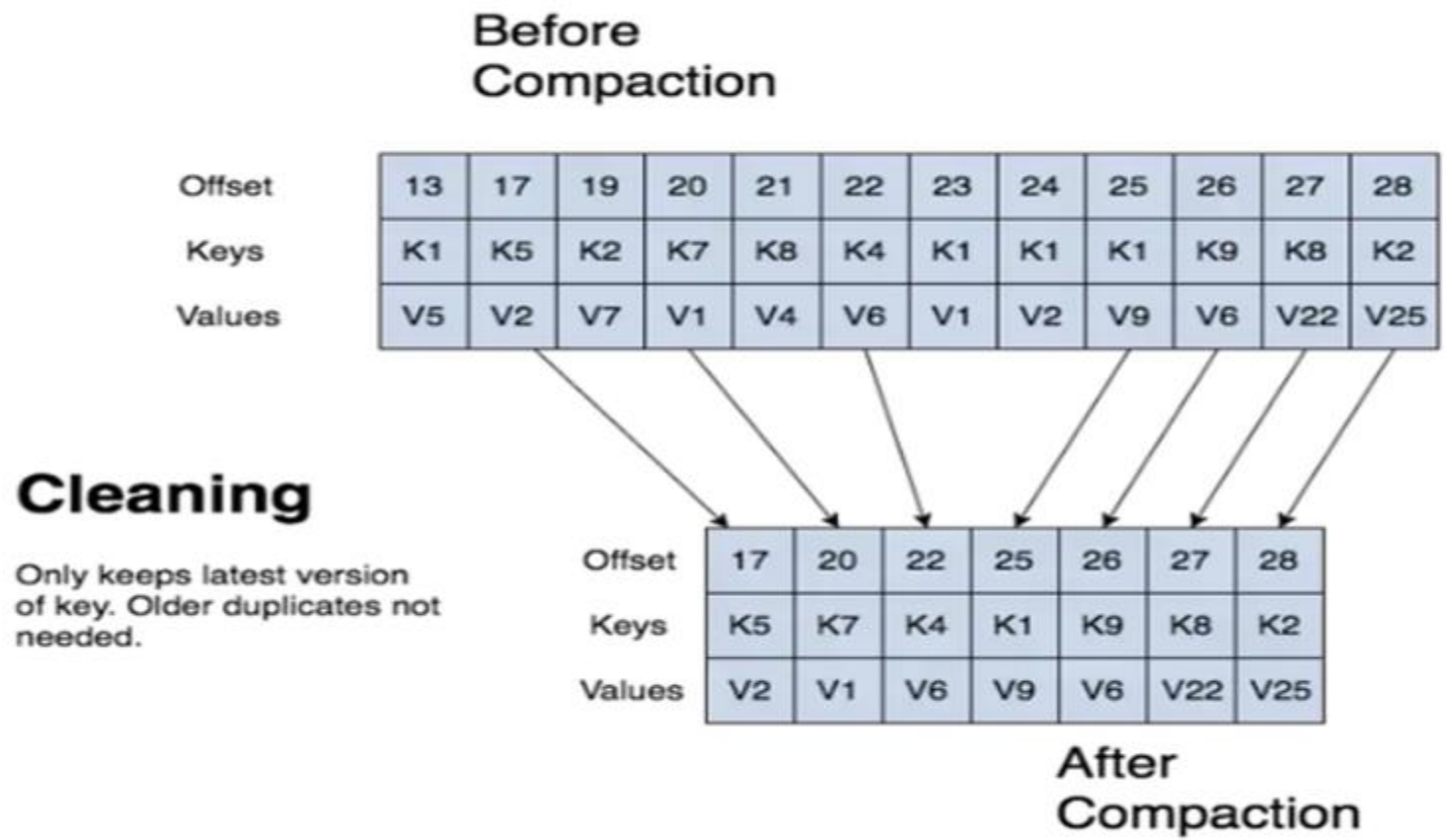
log.retention.bytes
max size in bytes for each partition [defailt is infinite (-1)]

Realtime usecases:

1. One week of retention:
   log.retention.hours=168 and log.retention.bytes=-1

2. Infinite time retention bounded by 500MB
   log.retention.hours=19999 and log.retention.bytes=524288000

Kafka Log Compaction Process



Producers, Consumers & Consumer Groups

Producers

- write data to topics (i.e., partitions)
- Producer automatically know to which broker and partition to write to
- In case of Broker failure, producers will automatically recover

Producers can choose to receive acknowledgement of data writes:

- acks=0: Producer won't wait for acknowledgment (chances of loosing the data)
- acks=1: Producer will wait for leader acknowledgment (limited data loss)
- acks=all : Leader + replicas acknowledgment (no data loss)

Producers : Message Keys

- Producers can choose to send a key with message (string, number etc.,)
- If key=null, data is sent round robin to brokers
- If a key is set, then all messages for that key will always go to the same partition
- A key is basically sent if you need message ordering for a specific field (ex: truck_id)

Consumers

- ❑ consumers read data from a topic
- ❑ consumers know which broker to read from
- ❑ In case of broker failures, consumers know how to recover
- ❑ Data is read in order within each partitions

Consumer Groups

- consumers read data in consumer groups
- each consumer within a group reads from exclusive partitions
- if you have more consumers than partitions, some consumers will be inactive

Delivery semantics for consumers

consumers choose when to commit offsets:

- ❑ At most once
- ❑ At least once (usually preferred)
- ❑ Exactly once (transactional)

At most once:

offsets are committed as soon as the message is received
if the processing goes wrong, the message will be lost

At least once:

offsets are committed after the message is processed if the processing goes wrong, the message will be read again this can result in duplicate processing of messages. use idempotent consumers.

Exactly once:

can be achieved for Kafka => kafka workflows using kafka streams api

At-most-once Kafka Consumer (Zero or more deliveries)
At-most-once consumer is the default behavior of a KAFKA consumer.

To configure this type of consumer,

- ❑ Set ‘enable.auto.commit’ to true
- ❑ Set ‘auto.commit.interval.ms’ to a lower timeframe.
- ❑ And do not make call to consumer.commitSync(); from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

At-Least-Once Kafka Consumer (One or more message deliveries, duplicate possible)

To configure this type of consumer:

- ❑ Set ‘enable.auto.commit’ to true
- ❑ Set ‘auto.commit.interval.ms’ to a higher number.
- ❑ Consumer should then take control of the message offset commits to Kafka by making the following call consumer.commitSync()

Kafka __consumer_offsets topic

Since kafka 0.9, it's not zookeeper anymore that store the information about what were the offset consumed by each groupid on a topic by partition.

Kafka now store this information on a topic called __consumer_offsets

As it is binary data, to see what's inside this topic we will need to consume with the formatter OffsetsMessageFormatter :

For kafka 0.11.0.0 and above

LET'S PRACTICE

```
$kafka-console-consumer.sh --formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsM
essageFormatter" --bootstrap-server kafka:9092 --topic
__consumer_offsets
```

The output will have the below format:

[groupId,topicName,partitionNumber]::[OffsetMetadata[OffsetNumber,NO_METADATA],CommitTime 1520613132835,ExpirationTime 1520699532835]

[test-group,test-topic,0]::[OffsetMetadata[20,NO_METADATA],CommitTime 1520613211176,ExpirationTime 1520699611176]

* for test-topic on test-group, consumer read to the offset 20, on the partition 0.

But maybe I have 30 offsets to be read on my partition 0, and it should alert me, that the worker that should be reading the partition 0, is late or for some reason are not committing the offsets as read, so potentially an error should be fixed.

For that there are another command that can help us (you should know the name of the groupid you want to monitor):

LET'S PRACTICE

$kafka-run-class.sh kafka.admin.ConsumerGroupCommand --bootstrap-server kafka:9092 --group my-group --describe

Topic	Partition	Current-offset	log-end-offset	Lag
test-topic	0	20	20	0
test-topic	1	-	1	-

if they are equal it’s perfect, - means that we don’t have yet the information on the __consumer_offsets.

Cluster Membership

Kafka uses Apache Zookeeper to maintain the list of brokers that are currently members of a cluster.

Every broker has a unique identifier that is either set in the broker configuration file or automatically generated.

Every time a broker process starts, it registers itself with its ID in Zookeeper by creating an ephemeral node.

Different Kafka components subscribe to the ***/brokers/ids*** path in Zookeeper where brokers are registered so they get notified when brokers are added or removed.

When a broker loses connectivity to Zookeeper, the ephemeral node that the broker created when starting will be automatically removed from Zookeeper.

Kafka components that are watching the list of brokers will be notified that the broker is gone.

Even though the node representing the broker is gone when the broker is stopped, the **broker ID still exists** in other data structures.

Start a brand **new broker with the ID of the old one**, it will immediately join the cluster in place of the missing broker with the same partitions and topics assigned to it.

The Controller

The controller is one of the Kafka brokers that, in addition to the usual broker functionality, is responsible for electing partition leaders.

The first broker that starts in the cluster becomes the controller by creating an ephemeral node in ZooKeeper called /controller.

The other brokers create a Zookeeper watch on the controller node so they get notified of changes to this node.

When the controller broker is stopped or loses connectivity to Zookeeper, the ephemeral node will disappear.

Other brokers in the cluster will be notified through the Zookeeper watch that the controller is gone and will attempt to create the controller node in Zookeeper themselves.

The first node to create the new controller in Zookeeper is the new controller, while the other nodes will receive a "node already exists" exception and re-create the watch on the new controller node.

When the controller notices that a broker left the cluster (by watching the relevant Zookeeper path), it knows that all the partitions that had a leader on that broker will need a new leader.

It goes over all the partitions that need a new leader, determines who the new leader should be (simply the next replica in the replica list of that partition), and sends a request to all the brokers that contain either the new leaders or the existing followers for those partitions.

Out-of-Sync Replicas

Seeing one or more replicas rapidly flip between in-sync and out-of-sync status is a sure sign that something is wrong with the cluster.

The cause is often a misconfiguration of Java's garbage collection on a broker. Misconfigured garbage collection can cause the broker to pause for a few seconds, during which it will lose connectivity to Zookeeper.

When a broker loses connectivity to Zookeeper, it is considered out-of-sync with the cluster.

Replication Factor (default is 1)

The topic-level configuration is replication.factor. At the broker level, you control the default.replication.factor for automatically created topics.

A reasonable assumption, topics had a replication factor of three, meaning that each partition is replicated three times on three different brokers

A replication factor of N allows you to lose N-1 brokers while still being able to read and write data to the topic reliably. So a higher replication factor leads to higher availability, higher reliability, and fewer disasters.

On the flip side, for a replication factor of N, you will need at least N brokers and you will store N copies of the data, meaning we will need N times as much disk space.

How do you determine the right number of replicas for a topic?

The answer is based on how critical a topic is and how much you are willing to pay for higher availability.

Placement of Replicas

By default, Kafka will make sure each replica for a partition is on a separate broker.

However, in some cases, this is not safe enough. If all replicas for a partition are placed on brokers that are on the same rack and the top-of-rack switch misbehaves, you will lose availability of the partition regardless of the replication factor.

To protect against rack-level misfortune, it is recommend placing brokers in multiple racks and using the broker.rack broker configuration parameter to configure the rack name for each broker.

broker.rack=my-rack-id

Unclean Leader Election

This configuration is only available at the broker (and in practice, cluster-wide) level. The parameter name is unclean.leader.election.enable and by default it is set to true.

When the leader for a partition is no longer available, one of the in-sync replicas will be chosen as the new leader. This leader election is “clean” in the sense that it guarantees no loss of committed data

when no in-sync replica exists except for the leader that just became unavailable?

This situation can happen in one of two scenarios:

1). The partition had three replicas, and the two followers became unavailable, In this situation, as producers continue writing to the leader, all the messages are acknowledged and committed.

Suppose, that the leader becomes unavailable??
In this scenario, if one of the out-of-sync followers starts first, we have an out-of-sync replica as the only available replica for the partition.

2). The partition had three replicas and, due to network issues, the two followers fell behind so that even though they are up and replicating, they are no longer in sync. The leader keeps accepting messages as the only in-sync replica.

Now if the leader becomes unavailable, the two available replicas are no longer in-sync.

In both these scenarios, we need to make a difficult decision:

1). If we don't allow the out-of-sync replica to become the new leader, the partition will remain offline until we bring the old leader (and the last in-sync replica) back online.

In some cases (e.g., memory chip needs replacement), this can take many hours.

2).If we do allow the out-of-sync replica to become the new leader, we are going to lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers.

Imagine that while replicas 0 and 1 were not available, we wrote messages with offsets 100-200 to replica 2 (then the leader).

Now replica 2 is unavailable and replica 0 is back online.

Replica 0 only has messages 0-100 but not 100-200. If we allow replica 0 to become the new leader, it will allow producers to write new messages and allow consumers to read them.

So, now the new leader has completely new messages 100-200. First, let's note that some consumers may have read the old messages 100-200, some consumers got the new 100-200, and some got a mix of both.

This can lead to pretty bad consequences when looking at things like downstream reports.

Setting **unclean.leader.election.enable** to true means we allow out-of-sync replicas to become leaders (knowns as unclean election), knowing that we will lose messages when this occurs. If we set it to false, we choose to wait for the original leader to come back online, resulting in lower availability

Minimum In-Sync Replicas

Both the topic and the broker-level configuration are called min.insync.replicas.

If you would like to be sure that committed data is written to more than one replica, We need to set the minimum number of in-sync replicas to a higher value.

If a topic has three replicas and you set min.insync.replicas to 2, then you can only write to a partition in the topic if at least two out of the three replicas are in-sync.

When all three replicas are in-sync, everything proceeds normally. This is also true if one of the replicas becomes unavailable.

However, if two out of three replicas are not available, the brokers will no longer accept produce requests. Instead, producers that attempt to send data will receive NotEnoughReplicasException.

Consumers can continue reading existing data. In effect, with this configuration, a single in-sync replica becomes read-only.

This prevents the undesirable situation where data is produced and consumed, only to disappear when unclean election occurs.

In order to recover from this read-only situation, we must make one of the two unavailable partitions available again (maybe restart the broker) and wait for it to catch up and get in-sync.