KIP-354: Add a Maximum Log Compaction Lag

- Status
- Motivation
    - Example
- Current Behavior
- Proposed Changes
- Public Interfaces
- Compatibility, Deprecation, and Migration Plan
- Performance impact
- Rejected Alternatives

Status

Current state: *Accepted*

Discussion thread: *[DISCUSS] KIP-354 Time-based log compaction policy*

Vote thread: [VOTE] KIP-354 Time-based log compaction policy

JIRA: KAFKA-7321

Pull Request: pull-6009

Motivation

Compaction enables Kafka to remove old messages that are flagged for deletion while other messages can be retained for a relatively longer time. Today, a log segment may remain un-compacted for an unbound time since the eligibility for log compaction is determined based on dirty ratio (“min.cleanable.dirty.ratio”) and min compaction lag ("min.compaction.lag.ms") setting. Ability to delete a record through compaction in a timely manner has become an important requirement in some use cases (e.g., GDPR). For example, one use case is to delete PII (Personal Identifiable information) data within certain days (e.g., 7 days) while keeping non-PII indefinitely in compacted format. The goal of this change is to provide a configurable maximum compaction lag that ensures a record is compacted after the specified time interval.

Example

```
A compacted topic with user id as key and PII in the value:


1 => {name: "John Doe", phone: "5555555"}
2 => {name: "Jane Doe", phone: "6666666"}

# to remove the phone number we can replace the value with a new message
1 => {name: "John Doe"}

# to completely delete key 1 we can send a tombstone record
1 => null

# but until compaction runs (and some other conditions are met), reading the whole topic will get all three
values for key 1, and the old values are still retained on disk.
```

This example mentions GDPR because it is widely known, but the requirement here is to provide some guarantees around a tombstone or a new value leading to deletion of old values within a maximum time.

Note: This Change focuses on when to compact a log segment, and it doesn’t conflict with KIP-280, which focuses on how to compact log.

Current Behavior

For log compaction enabled topic, Kafka today uses “min.cleanable.dirty.ratio” and "min.compaction.lag.ms" to determine what log segments it needs to pick up for compaction. "min.compaction.lag.ms" marks a log segment uncleanable until the segment is rolled and remains un-compacted for the specified "lag". The detailed information can be found in KIP-58. In addition, only log partitions whose dirty ratio is larger than “min.cleanable.dirty.ratio” are picked up by log cleaner for compaction. In summary, with these two existing compaction configurations, Kafka cannot enforce a maximum lag on compacting an un-compacted message record.

Proposed Changes

We propose adding a new topic level configuration: “max.compaction.lag.ms”, which controls the max lag after which a record is required to be picked up for compaction (note that this lag interval includes the time the record resides in an active segment). The lag is measured starting from when the message record is appended to an active segment. Since we reply on message timestamp to tell when a should be compacted, the feature provided in this KIP depends on the availability of message timestamp.

Here are a list of changes to enforce such a max compaction lag:

1. Force a roll of non-empty active segment if the first record is older than "max.compaction.lag.ms" so that compaction can be done on that segment. The time to roll an active segments is controlled by "segment.ms" today. However, to ensure messages currently in the active segment can be compacted in time, we need to roll the active segment when either "max.compaction.lag.ms" or "segment.ms" is reached.
We define:

   maximum time to roll an active segment:

   ```
   maxSegmentMs =  if (the log has "compact" enabled) {min(“segment.ms”, “max.compaction.lag.ms")}
                   else {“segment.ms” }
   ```

2. Estimate the earliest message timestamp of an un-compacted log segment. we only need to estimate earliest message timestamp for un-compacted log segments to ensure timely compaction because the deletion requests that belong to compacted segments have already been processed. The estimated earliest timestamp of a log segment is set to the timestamp of the first message. The message timestamp can be out of ordered in a log segment. However, when combining “max.compaction.lag.ms" with "log.message.timestamp.difference.max.ms", Kafka can provide actual guarantee that a new record will be eligible for log compaction after a bounded time as determined by “max.compaction.lag.ms" and "log.message.timestamp.difference.max.ms"[1].
3. Let log cleaner pick up logs that have reached max compaction lag for compaction.
For any given log partition with compaction enabled, as long as the estimated earliest message timestamp of first un-compacted segment is earlier than "max.compaction.lag.ms", the log is picked up for compaction. Otherwise, Kafka uses "min.cleanable.dirty.ratio" and "min.compaction.lag.ms" to determine the log's eligibility for compaction as it does today.

4. Add one Metric to track the max compaction delay (as described in the next section)

Public Interfaces

- Adding topic level configuration "max.compaction.lag.ms", and corresponding broker configuration "log.cleaner.max.compaction.lag.ms", which is set to MAX_LONG by default. Kafka validates "max.compaction.lag.ms" is no less than "min.compaction.lag.ms". A record may remain un-compacted for this max lag, after which the corresponding log partition becomes eligible for log compaction. This configuration only applies to topics that have compaction enabled.
- Add the following metric:

  1) kafka.log:type=LogCleaner,name=max-compaction-delay-secs
  type: gauge
  value: Math.max(now - earliest_timestamp_of_uncompacted_segment - max.compaction.lag.ms, 0)/1000
  This value is calculated across all compact-able partitions, where the max.compaction.lag.ms can be overridden on per-topic basis.

Compatibility, Deprecation, and Migration Plan

- By default "max.compaction.lag.ms" is set to MAX_LONG and this max compaction lag rule will not lead to additional log compaction. There are no compatibility issues and no migration is required.

Performance impact

- If a log partition already gets compacted once per day before this KIP, setting the log compaction time interval to more than one day should have little impact on the amount of resource spent on compaction since the existing log compaction configuration (e.g., min dirty ratio) will trigger log compaction before "max.compaction.lag.ms".   The added metric "max-compaction-delay-secs" can be used to determine whether there are some partitions are actually determined by "max.compaction.lag.ms" to be compacted.

Rejected Alternatives

- One way to force compaction on any cleanable log segment is setting “min.cleanable.dirty.ratio” to 0. However, compacting a log partition whenever a segment become cleanable (controlled by "min.compaction.lag.ms") is very expensive. We still want to accumulate some amount of log segments before compaction is kicked out. In addition, in order to honor the max compaction lag requirement, we also need to force a roll on active segment if the required lag has passed. So the existing configuration doesn't meet requirements to ensure a maximum compaction lag.
- In Item 2 of the proposed change section, if first message timestamp is not available, we use "segment.largestTimestamp - maxSegmentMs” as an estimation of earliest timestamp. The actual timestamp of the first message might be later than the estimation, but it is safe to pick up the log for compaction earlier. However, since this estimation is not very accurate and may cause unnecessary compaction, we decide to make this feature depends on the availability of first message timestamp.

- In Item 2 of the proposed change section, use the largestTimestamp of previous segment as an estimation of next segment's earliest timestamp. Since the estimation may not be very accurate, we decide to keep it simple and always use the first message timestamp as an estimation of a log segment's earliest message timestamp.

[1] Assuming a user sets "max.compaction.lag.ms" to M and "log.message.timestamp.difference.max.ms" to D, and the current time is "now",   in the worst case, the first message can have timestamp = (D + now), and the second message can have timestamp = (now - D).   This segment will become eligible for compaction at time (D+now+M). The compaction delay for the second message is (D+M+D). If we do have a huge timestamp shift between messages, the record is still bounded by (D+M+D) to become eligible for compaction. In general, if we don't expect huge timestamp shift, we can rely on "max.compaction.lag.ms" alone to trigger a compaction after the max lag.