KIP-534: Retain tombstones and transaction markers for approximately delete.retention.ms milliseconds

Status

Current state: Adopted

Discussion thread: *Thread*

JIRA: *KAFKA-8522*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Note: A pull request has been created for this KIP. (See here: PR#7175)

Contributions have been made to this KIP by Jun Rao and Jason Gustafson.

Problem

The idea of the configuration delete.retention.ms for compacted topics is to prevent an application that has read a key to not see a subsequent deletion of the key because it's physically removed too early. To solve this problem, from the latest possible time (deleteHorizonMs) that an application could have read a non tombstone key before a tombstone, we preserve that tombstone for at least delete.retention.ms and require the application to complete the reading of the tombstone by then.

deleteHorizonMs is no later than the time when the cleaner has cleaned up to the tombstone. After that time, no application can read a non-tombstone key before the tombstone because they have all been cleaned away through compaction. Since currently we don't explicitly store the time when a round of cleaning completes, deleteHorizonMs is estimated by the last modified time of the segment containing firstDirtyOffset. When merging multiple log segments into a single one, the last modified time is inherited from the last merged segment. So the last modified time of the newly merged segment is actually not an accurate estimate of deleteHorizonMs. It could be arbitrarily before (KAFKA-4545 <https://issues.apache.org/jira/browse/KAFKA-4545>) or after (KAFKA-8522 <https://issues.apache.org/jira/browse/KAFKA-8522>). The former causes the tombstone to be deleted too early, which can cause an application to miss the deletion of a key. The latter causes the tombstone to be retained longer than needed and potentially forever.

This issue also not only applies to tombstones, but the deletion of transaction markers as well. This KIP intends to resolve both issues.

Proposed Approach

In the version 2 format for batch headers, these are the contents that are stored in the headers: the base timestamp and the max timestamp. Each record in the batch contains a timestamp delta which is relative to the base timestamp. In other words, to get the record timestamp, you add the record delta to the base timestamp. Typically there is no reason for the base timestamp to be different from the timestamp of the first message, but this is not necessarily an iron rule. We can still retrieve the record timestamp if the delta and the base timestamp adds up to it. So the idea is to set the base timestamp to the delete horizon and adjust the deltas accordingly.

This might be a problem if the user are sticking with an older version of the batch header, but we can just add some documentation to note the limitation of this fix. (Most users would have migrated to v2 batch headers anyways). We can set a magic bit (as a delete horizon flag) in batch attributes to indicate if the batch header's timestamp and delta was modified. So in theory, when cleaning the log, it would look something like this:

Case 1: Normal batch

a. If the delete horizon flag is set, then retain tombstones as long as the current time is before the horizon.
b. If no delete horizon is set, then retain tombstones and set the delete horizon in the cleaned batch to current time + log.cleaner.delete.retention.ms.

Case 2: Control batch

a. If the delete horizon flag is set, then retain the batch and the marker as long as the current time is before the horizon.
b. If no delete horizon is set and there are no records remaining from the transaction, then retain the marker and set the delete horizon in the cleaned batch to current time + log.cleaner.delete.retention.ms.

For the magic bit, the old version of the batch attribute is the following:

| Unused (6-15) | Control (5) | Transactional (4) | Timestamp Type (3) | Compression Type (0-2) |

We will set bit number 6 as the delete horizon flag, in which case, the batch attribute becomes:

| Unused (7-15) | Delete Horizon Flag (6) | Control (5) | Transactional (4) | Timestamp Type (3) | Compression Type (0-2) |

There is the possibility that on large batch deletions, we could exceed the max message size. This could likely occur when the deltas are small negative values (i.e. -1 for example) in which case we would take more bits to encode them. While this would not happen under most circumstances, it is worth noting.

Rejected Alternatives

Below are the rejected proposals that we have to date. These approaches can only fix the delete horizon issue for tombstones. The deletion of transaction markers however will still not be fixed. These approaches were abandoned due to their inability to fix the second issue we are trying to tackle.

Proposal 1

Note: This proposal has been rejected since this approach does not cover transaction markers.

We could store this cleaning time in a checkpoint file (which is used to store both the end offsets of cleaned segments as well as the time at which this cleaning occurred). In this manner, when the checkpoint file is queried for the last offset checkpointed, the last cleaned time can be retrieved with it and be used to calculate the segment's delete horizon.

The old format of the checkpoint file is more compact, using one file per disk. This has a couple of implications. One file would store end offsets from multiple partitions, which meant that a typical file in the old system would look like this:

Old Organization

```
(PARTITION1, OFFSET1)
(PARTITION2, OFFSET2)
...
(PARTITIONK, OFFSETM)
```

In the new checkpoint file systen, we propose to have each partition be assigned their own specific checkpoint file–which means multiple files would exist on one disk (this is usually the case since the number of partitions exceed the number of disks in most use cases). Since we wish to store offset times as well, we will checkpoint this information in a tuple consisting of an end offset and its respective cleaning time. Therefore, a file would now look like the following:

New File System

```
(END_OFFSET, CLEANING_TIME)
```

Upgrade Path For Proposal 1

It is possible that when the user upgrades Kafka to a newer version, information could still be stored in the old checkpoint files. In which case, an upgrade path must be provided.

When a new log cleaner instance is created, we propose to read any remaining data that is stored in the old checkpoint files and rewrite the offsets we have read into the new file system we implemented. Any operations that are then carried out will no longer use the old checkpoint file system but the new one.

Proposal 2

Note: This proposal has a possibility of exposing delete horizon information to the client, which should be avoided.

Another option is to store an individual's tombstone deleteHorizonMs in an internal header field of the record. In this case, if the tombstone's offset is before the first dirty offset in a log, it would be used to determine if it should be removed whenever a log cleaner thread scans over it. This has some pluses over the first proposal because there is no longer a need to change the file system and in theory should be simpler to implement.

The upgrade path for this alternative is relatively simple. All we have to do is check for the existence of this new field in tombstones (and it does not exist in older versions). If it doesn't exist, then we can default back to the old system of using the last modified time to calculate the deleteHorizonMs.