KIP-107: Add deleteRecordsBefore() API in AdminClient

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *here*

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

Motivation

Kafka can be used in a stream processing pipeline to pass intermediate data between processing jobs. The amount of intermediate data generated from stream processing jobs can taken a large amount of disk space in the Kafka. It is important that we can delete this data soon after it is consumed by downstream application, otherwise we have to pay significant cost to purchase disks for Kafka clusters to keep those data.

However, Kafka doesn't provide any mechanism to delete data after data is consumed by downstream jobs. It provides only time-based and size-based log retention policy, both of which are agnostic to consumer's behavior. If we set small time-based log retention for intermediate data, the data may be deleted even before it is consumed by downstream jobs. If we set large time-based log retention, the data will take large amount of disk space for a long time. Neither solution is good for Kafka users. To address this problem, we propose to add a new admin API which can be called by user to delete data that is no longer needed.

Note that this KIP is related to and supersedes KIP-47.

Public Interfaces

1) Java API

- Add the following API in Admin Client. This API returns a future object whose result will be available within RequestTimeoutMs, which is configured when user constructs the AdminClient.

```
Future<Map<TopicPartition, DeleteDataResult>> deleteRecordsBefore(Map<TopicPartition, Long> offsetForPartition)
```

- DeleteDataResult has the following two fields, which tells user if the data has been successfully deleted for the corresponding partition.

```
DeleteDataResult(long: low_watermark, error: Exception)
```

2) Protocol

Create DeleteRecordsRequest

PurgeRequest

```
DeleteRecordsRequest => topics timeout
  topics => [DeleteRecordsRequestTopic]
  timeout => int32

DeleteRecordsRequestTopic => topic partitions
  topic => str
  partitions => [DeleteRecordsRequestPartition]

DeleteRecordsRequestPartition => partition offset
  partition => int32
  offset => int64  // offset -1L will be translated into high_watermark of the partition when leader receives
the request.
```

Create DeleteRecordsResponse

PurgeReponse

```
DeleteRecordsResponse => topics
  topics => [DeleteRecordsResponseTopic]

DeleteRecordsResponseTopic => topic partitions
  topic => str
  partitions => [DeleteRecordsResponsePartition]

DeleteRecordsResponsePartition => partition low_watermark error_code
  partition => int32
  low_watermark => int64
  error_code => int16
```

Add a log_start_offset field to FetchRequestPartition

FetchRequestPartition

```
FetchRequestPartition => partition fetch_offset low_watermark max_bytes
  partition => int32
  fetch_offset => int64
  log_start_offset => int64  <-- NEW. If it is issued from consumer, the value is 0. Otherwise, this is the
log_start_offset of this partition on the follower.
  max_bytes => int32
```

Add a log_start_offset field to FetchResponsePartitionHeader

FetchResponsePartitionHeader

```
FetchResponsePartitionHeader => partition error_code high_watermark low_watermark
  partition => int32
  error_code => int16
  high_watermark => int64
  log_start_offset => int64  <-- NEW. This is the low_watermark of this partition on the leader.
```

3) Checkpoint file

We create one more checkpoint file, named "log-begin-offset-checkpoint", in every log directory. The checkpoint file will have the same format as existing checkpoint files (e.g. replication-offset-checkpoint) which map TopicPartition to Long.

4) Script

Add kafka-delete-data.sh that allows user to delete data in the command line. The script requires for the following arguments:

- bootstrap-server. This config is required from user. It is used to identify the Kafka cluster.
- command-config. This is an optional property file containing configs to be passed to Admin Client.
- delete-offset-json-file. This config is required from user. It allows user to specify offsets of partitions to be delete. The file has the following format:

```
{
  "version" : int,
  "partitions" : [
    {
      "topic": str,
      "partition": int,
      "offset": long
    },
    ...
  ]
}
```

Proposed Changes

The idea is to add new APIs in Admin Client (see KIP-4) that can be called by user to delete data that is no longer needed. New request and response needs to be added to communicate this request between client and broker. Given the impact of this API on the data, the API should be protected by Kafka's authorization mechanism described in KIP-11 to prevent malicious or unintended data deletion. Furthermore, we adopt the soft delete approach because it is expensive to delete data in the middle of a segment. Those segments whose maximum offset < offset-to-delete can be deleted safely. Brokers can increment log_start_offset of a partition to offset-to-delete so that data with offset < offset-to-delete will not be exposed to consumer even if it is still on the disk. And the log_start_offset will be checkpointed periodically similar to high_watermark to be persistent.

Note that the way broker handles DeleteRecordsRequest is similar to how it handles `ProduceRequest` with ack = all and isr=all_live_replicas, e.g. the leader waits for all followers to catch up with its `log_start_offset`, doesn't expose message below `log_start_offset`, and checkpoints `log_start_offset` periodically. The `low_watermark` of a partition will be the minimum `log_start_offset` of all replicas of this partition and this value will be returned to user in DeleteRecordsResponse.

Please refer to public interface section for our design of the API, request and response. In this section we will describe how broker maintains low watermark per partition, how client communicates with broker to delete old data, and how this API can be protected by authorization.

1) Interaction between user application and brokers

1) User application determines the maximum offset of data that can be deleted per partition. This information is provided to `deleteRecordsBefore()` as Map<TopicPartition, Long>. If users application only knows timestamp of data that can be deleted per partition, they can use `offsetsForTimes()` API to convert the cutoff timestamp into offsetToDelete per partition before providing the map to deleteRecordsBefore() API.

2) Admin Client builds DeleteRecordsRequest using the offsetToDelete from `deleteRecordsBefore`() parameter and the `requestTimeoutMs` is taken from the `AdminClient` constructor. One `DeleteRecordsRequest` is sent to each broker that acts as leader of any partition in the request. The request should only include partitions which the broker leads.

3) After receiving the `DeleteRecordsRequest`, for each partition in the `DeleteRecordsRequest`, the leader first sets offsetToDelete to `high_watermark` if offsetToDelete is -1L. It then sets `log_start_offset` of leader replica to `max`(`log_start_offset`, offsetToDelete) if offsetToDelete <= `high_watermark`. Those segments whose largest offset < `log_start_offset` will be deleted by the leader.

4) The leader puts the `DeleteRecordsRequest` into a `DelayedOperationPurgatory`. The `DeleteRecordsRequest` can be completed when results for all partitions specified in the DeleteRecordsRequest are available. The result of a partition will be available within RequestTimeoutMs and it is determined using the following logic:

- If `log_start_offset` of this partition on all live followers is larger than or equal to the offsetToDelete, the result of this partition will be its `low_watermark`, which is the minimum `log_start_offset` of all its live replicas.
- If `high_watermark` of this partition is smaller than the offsetToDelete, the result of this partition will be `OffsetOutOfRangeException`.
- If the leadership of this partition moves to another broker, the result of this partition will be `NotLeaderException`
- If the result of this partition is not available after `RequestTimeoutMs`, the result of this partition will be `TimeoutException`

5) The leader sends `FetchResponse` with its `log_start_offset` to followers.

6) Follower sets replica's `log_start_offset` to the max(`log_start_offset` of leader, `log_start_offset` of local replica). It also deletes those segments whose largest offset < `log_start_offset`.

7) Follower sends `FetchRequest` with replica's `log_start_offset` to the leader.

8) The leader updates `log_start_offset` of each follower. If the `DeleteRecordsRequest` can be completed, the leader removes the `DeleteRecordsRequest` from `DelayedOperationPurgatory` and sends `DeleteRecordsResponse` with the results (i.e. low_watermark or error) for the specified set of partitions.

9) If admin client does not receive `DeleteRecordsResponse` from a broker within `RequestTimeoutMs`, the `DeleteDataResult` of the partitions on that broker will be`DeleteDataResult`(low_watermark = -1, error = TimeoutException). Otherwise, the `DeleteDataResult` of each partition will be constructed using the `low_watermark` and the `error`of the corresponding partition which is read from the `DeleteDataResponse` received from its leader broker. deleteRecordsBefore`(...).get()` will unblock and return `Map<TopicPartition, DeleteDataResult>` when `DeleteDataResult` of all partitions specified in the `offsetForPartition` param are available.

2) Routine operation in the broker

- Broker will delete those segments whose largest offset < `log_start_offset`.

- Only message with offset >= `log_start_offset` can be sent to consumer.

- When a segment is deleted due to log retention, broker updates `log_start_offset` to max(`log_start_offset`, smallest offset in the replica's log)

- Broker will checkpoint `log_start_offset` for all replicas periodically in the file "log-begin-offset-checkpoint", in the same way it checkpoints `high_watermark` of replicas. The checkpoint file will have the same format as existing checkpoint files which map TopicPartition to Long.

3) API Authorization

Given the potential damage that can be caused if this API is used by mistake, it is important that we limit its usage to only authorized users. For this matter, we can take advantage of the existing authorization framework implemented in KIP-11. `deleteRecordsBefore()` will have the same authorization setting as `deleteTopic()`. Its operation type is be DELETE and its resource type is TOPIC.

4) ListOffsetRequest

`log_start_offset` of a partition will be used to decide the smallest offset of the partition that will be exposed to consumer. It will be returned when smallest_offset option is used in the ListOffsetRequest.

Compatibility, Deprecation, and Migration Plan

This KIP is a pure addition, so there is no backward compatibility concern.

The KIP changes the inter-broker protocol. Therefore the migration requires two rolling bounce. In the first rolling bounce we will deploy the new code but broker will still communicate using the existing protocol. In the second rolling bounce we will change the config so that broker will start to communicate with each other using the new protocol.

Test Plan

- Unit tests to validate that all the individual components work as expected.
- Integration tests to ensure that the feature works correctly end-to-end.

Rejected Alternatives

- Using committed offset instead of an extra API to trigger data delete operation. Delete data if its offset is smaller than committed offset of all consumer groups that need to consume from this partition.
This approach is discussed in KIP-68. The advantage of this approach is that it doesn't need coordination of user applications to determine when `deleteRecordsBefore`() can be called, which can be hard to do if there are multiple consumer groups interested in consuming this topic. The disadvantage of this approach is that it is less flexible than `deleteRecordsBefore`() API because it re-uses committed offset to trigger data delete operation. Also, it adds complexity to broker implementation and would be more complex to implement than the `deleteRecordsBefore`() API. An alternative approach is to implement this logic by running an external service which calls `deleteRecordsBefore`() API based on committed offset of consumer groups.

- Leader sends DeleteRecordsResponse without waiting for `low_watermark` of all followers to increase above the cutoff offset
This approach would be simpler to implement since it doesn't require DelayedOperationPurgatory for DeleteRecordsRequest. The leader can reply to DeleteRecordsRequest faster since it doesn't need to wait for followers. However, the deleteRecordsBefore() API would provide weaker guarantee in this

approach because the data may not be deleted if the leader crashes right after it sends DeleteRecordsResponse. It will be useful to know for sure whether the data has been deleted, e.g. when user wants to delete problematic data from upstream so that downstream application can re-consume clean data, or if user wants to delete some sensitive data.

- Delete data on only one partition by each call to deleteRecordsBefore(...)
This approach would make the implementation of this API simpler, and would be consistent with the existing seek(TopicPartition partition, long offset) API. The downside of this approach is that it either increases the time to delete data if the number of partitions is large, or it requires user to take extra effort to parallelize the deleteRecordsBefore(...). This API may take time longer than seek() for a given partition since the broker needs to wait for follower's action before responding to deleteDataRequest. Thus we allow user to specify a map of partitions to make this API easy to use.