KIP-253: Support in-order message delivery with partition expansion

Status

Current state: *Under Discussion*

Discussion thread:

JIRA:

Motivation

Kafka is designed to allow messages with the same key from the same producer to be consumed in the same order as they are produced. This feature is useful for applications which maintain local states per key. However, as of current design of Kafka, this in-order delivery is not guaranteed if we expand partition of the topic. Furthermore, for topic whose traffic fluctuate significantly over time, it will be useful to be able to expand partitions when the expected byte-in-rate is high and delete partitions when the expected byte-in-rate is low. This KIP proposes a design to allow partition expansion and deletion while still ensuring in-order message delivery for keyed messages.

Goals

This KIP allows arbitrary sequence of partition expansion and deletion of an existing topic while still ensuring in-order message delivery for keyed messages, except that user can not expand partitions of an existing topic when there is still partition marked for deletion for that topic. After the size of this partition reaches zero, due to either retention or AdminClient.deleteRecords(...), then this partition will be removed from the topic and user can expand partitions of this topic.

In the future we can expand upon the work of this KIP to support partition expansion even when there is still partition marked for deletion for that topic.

The current KIP w.r.t. the interface that our producer and consumer exposes to the user. It ensures that if there are two messages with the same key produced by the same producer, say messageA and messageB, and suppose messageB is produced after messageA to a different partition than messageA, then we can guarantee that the following sequence can happen in order:
- Consumer of messageA can execute callback, in which user can flush state related to the key of messageA.
- messageA is delivered by its consumer to the application
- Consumer of messageB can execute callback, in which user can load the state related to the key of messageB.
- messageB is delivered by its consumer to the application.

Public Interfaces

Zookeeper

1) Update the znodes /brokers/topics/[topic] to use the following json format

```
{
  "version" : int32,
  "partitions" : {
    partition -> replicaList
    ...
  },
  "initial_partition_count" : int32    <-- NEW. This is the number of partitions when the topic is created.
  "undeleted_partition_count" : int32  <-- NEW. This is the partition_count used by producer for choosing
partitions.
}
```

2) Update the znodes /brokers/topics/[topic]/partitions/[partition]/state to use the following json format

```
{
  "version" : int32,
  "partition_epoch" : int32
  "leaderEpochAfterCreation" : {      <-- NEW. This represents a map from partition to leaderEpoch for lower
partitions.
    int32 -> int32
    ...
  },
  "leaderEpochBeforeDeletion" : {     <-- NEW. This represents a map from partition to leaderEpoch for lower
partitions.
    int32 -> int32
    ...
  }
}
```

Protocol

1) Update LeaderAndIsrRequest to re-use topic field for all its partitions and add field undeleted_partition_count for each topic.

```
LeaderAndIsrRequest => controller_id controller_epoch topic_states live_leaders
  controller_id => int32
  controller_epoch => int32
  topic_states => [LeaderAndIsrRequestTopicState]    <-- NEW. This field includes
LeaderAndIsrRequestPartitionState
  live_leaders => [LeaderAndIsrRequestLiveLeader]

LeaderAndIsrRequestTopicState => topic partition_states
  topic => str                                       <-- This is moved from LeaderAndIsrRequestPartitionState.
  undeleted_partition_count => int32                 <-- NEW. This is the total number of partitons of this
topic.
  partition_states => [LeaderAndIsrRequestPartitionState]

LeaderAndIsrRequestPartitionState => partition leader leader_epoch isr zk_version replicas
  partition => int32
  controller_epoch => int32
  leader => int32
  leader_epoch => int32
  isr => [int32]
  zk_version => int32
  replicas => [int32]
  is_new_replica =>  boolean
```

2) Update ProduceRequest to include undeleted_partition_count per topic.

```
ProduceRequest => transactional_id acks timeout topic_data
  transaction_id => nullable_str
  acks => int16
  timeout => int32
  topic_data => [TopicProduceData]

TopicData => topic data
  topic => str
  undeleted_partition_count => int32    <-- NEW. This is the number of undeleted partitons of this topic
expected by the producer.
  data => PartitionData

PartitionData => partition record_set
  partition => int32
  record_set => Records
```

3) Add ConsumerGroupPositionRequest that allows a consumer to report its own position of partitions and query position of partitions of other consumers in its consumer group.

```
ConsumerGroupPositionRequest => group_id generation_id member_id topics
  group_id => str
  generation_id => int32
  member_id => str
  topics => [ConsumerGroupPositionRequestTopic]

ConsumerGroupPositionRequestTopic => topic partitions
  topic => str
  partitions => [ConsumerGroupPositionRequestPartition]

ConsumerGroupPositionRequestPartition => partition need_position position
  partition => int32
  need_position => boolean // If true, ConsumerGroupPositionResponse should include the position of this
partition of the group.
  position => int64        // Position of this partition of this consumer.
```

4) Update ConsumerGroupPositionResponse to include fields need_position and position for relevant partitions.

```
ConsumerGroupPositionResponse => throttle_time_ms topics error_code
  throttle_time_ms => int32
  topics => [ConsumerGroupPositionResponseTopic]
  error_code => int16

ConsumerGroupPositionResponseTopic => topic partitions
  topic => str
  partitions => [ConsumerGroupPositionResponsePartition]

ConsumerGroupPositionResponsePartition => partition need_position position
  partition => int32
  need_position => boolean // If true, ConsumerGroupPositionRequest should include the position of this
partition of this consumer
  position => int64        // Position of this partition of the group.
```

5) Add PartitionLeaderEpochsForPartitionsRequest and PartitionLeaderEpochsForPartitionsResponse. PartitionLeaderEpochsForPartitionsResponse essentially encodes the leaderEpochAfterCreation and the leaderEpochBeforeDeletion map for those partitions specified in the PartitionLeaderEpochsForPartitionsRequest.

```
PartitionLeaderEpochsForPartitionsRequest => topics
  topics => [PartitionLeaderEpochsForPartitionsRequestTopic]

PartitionLeaderEpochsForPartitionsRequestTopic => topic partitions
  topic => str
  partitions => [int32]
```

```
PartitionLeaderEpochsForPartitionsResponse => throttle_time_ms topics
  throttle_time_ms => int32
  topics => [PartitionLeaderEpochsForPartitionsResponseTopic]

PartitionLeaderEpochsForPartitionsResponseTopic => topic partitions
  topic => str
  partitions => [PartitionLeaderEpochsForPartitionsResponsePartition]

PartitionLeaderEpochsForPartitionsResponsePartition => partition leaderEpoch
  partition => int32
  leader_epoch_after_creation => int32   // -1 if the given partition is not in leaderEpochAfterCreation of the
partition znode.
  leader_epoch_before_deletion => int32  // -1 if the given partition is not in leaderEpochBeforeDeletion of
the partition znode.
```

6) Update UpdateMetadataRequest to re-use topic field for all its partitions and add fields undeleted_partition_count and initial_partition_count for each topic.

```
UpdateMetadataRequest => controller_id controller_epoch max_partition_epoch partition_states live_brokers
  controller_id => int32
  controller_epoch => int32
  max_partition_epoch => int32
  topic_states => [UpdateMetadataRequestTopicState]
  live_brokers => [UpdateMetadataRequestBroker]

UpdateMetadataRequestTopicState => topic partition_states
  topic => str
  initial_partition_count => int32     <-- NEW. This represents a map from partition to leaderEpoch for lower
partitions.
  undeleted_partition_count => int32    <-- NEW. This is the number of undeleted partitons of this topic
expected by the producer.
  partition_states => [UpdateMetadataRequestTopicState]

UpdateMetadataRequestPartitionState => partition controller_epoch leader leader_epoch partition_epoch isr
zk_version replicas offline_replicas
  partition => int32
  controller_epoch => int32
  leader => int32
  leader_epoch => int32
  partition_epoch => int32
  isr => [int32]
  zk_version => int32
  replicas => [int32]
  offline_replicas => [int32]
```

7) Add fields undeleted_partition_count and initial_partition_count for each topic in MetadataResponse.

```
MetadataResponse => throttle_time_ms max_partition_epoch brokers cluster_id controller_id topic_metadata
  throttle_time_ms => int32
  max_partition_epoch => int32
  brokers => [MetadatBroker]
  cluster_id => nullable_str
  controller_id => int32
  topic_metadata => [TopicMetadata]

TopicMetadata => topic_error_code topic is_internal partition_metadata
  topic_error_code => int16
  topic => str
  initial_partition_count => int32     <-- NEW. This represents a map from partition to leaderEpoch for lower
partitions.
  undeleted_partition_count => int32    <-- NEW. This is the number of undeleted partitons of this topic
expected by the producer.
  is_internal => boolean
  partition_metadata => [PartitionMetadata]

PartitionMetadata => partition_error_code partition_id leader replicas leader_epoch partition_epoch isr
offline_replicas
  partition_error_code => int16
  partition_id => int32
  leader => int32
  replicas => [int32]
  leader_epoch => int32
  partition_epoch => int32
  isr => [int32]
  offline_replicas => [int32]
```

8) Add field undeleted_partition_count for each topic in the FetchRequest

```
FetchRequest => replica_id max_wait_time min_bytes max_bytes isolation_level session_id epoch topics
forgetten_topics_data
  replica_id => int32
  max_wait_time => int32
  min_bytes => int32
  max_bytes => int32
  isolation_level => int8
  session_id => int32
  epoch => int32
  topics => [FetchRequestTopic]
  forgetten_topics_data => [ForgottenTopicData]FetchRequestTopic => topic undeleted_partition_count partitions
topic => str  undeleted_partition_count => int32    <-- NEW. This is the number of undeleted partitons of this
topic expected by the producer.
  partitions => [FetchRequestPartition]
```

Consumer API

1) Add the following method to the interface org.apache.kafka.clients.consumer.Consumer

```
public void subscribe(Collection<String> topics, ConsumerRebalanceListener consumerRebalanceListener,
PartitionKeyRebalanceListener partitionKeyRebalanceListener);


public interface PartitionKeyRebalanceListener {

  /*
   *  This callback allows user to flush state related to the keys previously received in the given set of
partitions before another consumer loads state for such keys and starts to consume messages with these keys.
This can happen after partition creation or deletion for an existing topic.
   */
  void onPartitionKeyMaybeRevoked(Collection<TopicPartition>);

  /*
   *  This callback allows user to load state for new keys that may be received in the given set of partitions.
This can happen after partition creation or deletion for an existing topic.
   */
  void onPartitionKeyAssigned(Collection<TopicPartition>);


}
```

Topic config

1) Add per-topic config enable.ordered.delivery. The default value is true. When it is set to false
When it is set to false, consumer will not performance the operation, described in Proposed Changes 4), 7) and 8), for partitions of the given topic.

Error code

1) Add a new exception InvalidPartitionMetadataException. This should be a retriable exception that extends InvalidMetadataException. It can be returned in ProduceResponse or FetchResponse.

Proposed Changes

1) Changes in the controller for handling topic creation and the existing topic znodes

- When the topic znode is created, controller should update the topic znode with the initial_partition_count. Controller should always write the initial_partition_count for existing topic znodes whose initial_partition_count is not specified.

initial_partition_count will be used by producer to map message key to partition.

2) Changes in the controller for handling partition expansion

- User executes kafka-topics.sh to update the topic znode with the new assignment. This triggers the topic znode listener in controller.

- Controller updates the topic znode with the new undeleted_partition_count.

- For those partitions of this topic which already have the partition znode, controller increments their leaderEpoch by 1 in the partition znode. Controller sends LeaderAndIsrRequest and wait for LeaderAndIsrResponse. The LeaderAndIsrRequest should include the new leaderEpoch for each partition and the updated undeleted_partition_count of the topic.

- For each partition of this topic which does not have the partition znode, controller creates the partition znode, such that the leaderEpochAfterCreation field in the znode data maps partition of this topic to the corresponding leaderEpoch (recorded before controller increments the leaderEpoch)

- Controller propagates the UpdateMetadataRequest with the latest undeleted_partition_count per topic.

- Controller continue the existing logic of partition expansion.

Note that this procedure is fault tolerant. If controller fails in any of these step, the new controller can continue creating partition znode following the same procedure.

The motivation of this change is that, for each existing partition before the partition expansion, there exists a leaderEpoch such that all messags after this leaderEpoch will be produced with the producers using the incremented undeleted_partition_count. If for all existing partitions, a consumer group has delivered all messages up to such leaderEpoch, then it should be safe to consume messages from the newly created partition without worrying about out-of-order delivery. Consumers can get such leaderEpoch from the partition znode, convert leaderEpoch to lowerOffsetThreshold using OffsetsForLeaderEpochRequest, and make sure that it only starts consuming new partition after messages before the lowerOffsetThreshold have all been consumed.

3) Changes in the controller for handling partition deletion

- User executes "kafka-topics.sh --alter --partition partitionNum --topic topic" to alter the partition count of the given topic. If the given undeleted_partition_count is smaller than the current undeleted_partition_count of the given topic AND no smaller than the initial_partition_count, the script writes the given undeleted_partition_count in the topic znode. This triggers the topic znode listener in controller.

- For those partitions P1 whose partitionId >= undeleted_partition_count, if the leaderEpochBeforeDeletion in the partition znode empty or it is not specified, controller updates partition znode of P1 such that for every undeleted partitions P2, leaderEpochBeforeDeletion maps the partition P2 to its current leaderEpoch. Controller also increments the leaderEpoch by 1 for partition znode of every P2. Then controller sends LeaderAndIsrRequest with the updated undeleted_partition_count for this topic and the updated leaderEpoch for P2.

- Controller propagates the UpdateMetadataRequest with the latest undeleted_partition_count per topic.

- Controlelr schedules a task that periodically which periodically sens ListOffsetRequest to check the size of those partitions which have been marked for deletion, i.e. with non-empty leaderEpochBeforeDeletion map in the parittion znode. If the size of a yet-to-be-deleted partition is 0, i.e. its logStartOffset == logEndOffset, controller sends StopReplicaRequest with delete=true, removes the partition from the assignment in the topic znode, and removes the partition znode.

Note that this procedure is fault tolerant. If controller fails in any of these step, the new controller can continue to read the undeleted_partition_count from the topic znode, update partition znode and sends LeaderAndIsrRequest.

The motivation of this change is that, for each undeleted partition after the partition deletion, there exists a leaderEpoch such that all messages up to this leaderEpoch are produced with the producers using the old undeleted_partition_count prior to the partition deletion. If the consumer group has delivered all messages in a partition that has been marked for deletion, it should be safe for this consumer group to consume any message in undeleted partition. If the consumer group has not delivered some messages in a partition that has been marked for deletion, it should still be safe for this consumer group to delivery messages up to the corresponding leaderEpoch in the undeleted partitions.

4) Changes in how broker handles ProduceRequest

- When broker receives LeaderAndIsrRequest, in addition to the existing procedure (e.g. updating the leaderEpochCache for the new leaderEpoch), the broker should also record in memory the undeleted_partition_count for each topic.

- When broker receives ProduceRequest, for each partition in the request, broker checks whether its undeleted_partition_count equals the undeleted_partition_count from the most recent LeaderAndIsrRequest. If yes, broker handles the produce request in the current way. If no, broker rejects this partition with InvalidPartitionMetadataException. This error extends InvalidMetadaException and should trigger producer to update its metadata and retry.

The motivation of this change is that all messags after the incremented leaderEpoch (read from the LeaderAndIsrRequest) will be produced by producers using the latest undeleted_partition_count (also read from the same LeaderAndIsrRequest). Note that the leaderEpoch and the undeleted_partition_count are updated atomically using the same LeaderAndIsrRequest.

5) Changes in how producer constructs ProduceRequest

- Producer should include the undeleted_partition_count for each topic in the ProduceRequest.

- Producer should determine the partition for both keyed and unkeyed message using Linear Hashing (see https://en.wikipedia.org/wiki/Linear_hashing), where N = initial_partition_count, L is the largest integer that satisfies N*2^L <= undeleted_partition_count, and S can similariy be derived using N, L and undeleted_partition_count.

- Producer will update metadata and retry ProduceRequest if ProduceResponse shows InvalidPartitionMetadataException, which happens if producer's undeleted_partition_count is different (maybe newer or older) than the undeleted_partition_count in the broker.

The motivation of these changes are:

- Producer always use the undeleted_partition_count in the leader broker to determine the partition of each message
- The benefits of using linear hashing as compared to existing hashing algorithm include:
  - Each newly added partition will be blocked on exactly one existing partition before it can be consumed.
  - Each newly deleted partition will block the consumption of at most O(log(n)) remaing partitions.
  - There is no movement of keys between existing partitions so that the existing partitions do not block on each other.
  - The movement of keys between partitions is much less and thus less state needs to be loaded/stored for stream processing.

Note: the extra steps introduced in here is skipped for partitions of topics whose enable.ordered.delivery is set to false.

6) Changes in the leader of the consumer group

- Leader of the consumer group splits the partition list to those partitions that have not been marked for deletion and those partitions that have been marked for deletion. It should apply the user-defined assignment algorithm to these two lists seperately to determine the partition distribution across consumers in the group, so that partitions which have not been marked for deletion can also be evenly distributed across consumers. This is to prevent load imbalance across consumers because there will be no new data to those partitions which have been marked for deletion.

The motivation of this change is to evenly distribute the byte-in-rate of undeleted partitions across consumers of the consumer group. Note that even though there is still remaning data in the partitions which have been marked for deletion, there is no new data into these partitions. Thus a consumer will likely be under-utilized if it is assigned few undeleted partitions but many partitons which have been marked for deletion.

7) Changes in how consumer queries position of partitions of the consumer group using ConsumerGroupPositionRequest and ConsumerGroupPositionResponse

- Periodically, after every heartbeat.interval.ms, consumer sends ConsumerGroupPositionRequest to the consumer group coordinator.

- ConsumerGroupPositionRequest includes the current position for each partition requested by the coordinator from the previous ConsumerGroupPositionResponse. It also includes the list of partitions for which it wants to know the position (of the consumer that is consuming this partitoin). See "Changes in how consumer consumes partition" below for how consumer determines the list of partitions for which it wants to know the position.

- Group coordinator keeps a map that maps the partition to the the list of consumer ids that are interested in the position of this partition. Group coordinator also remembers the positions for those partitions which are interesting to some consumers of the given group. Both information may be updated when coordinator receives ConsumerGroupPositionRequest.

- ConsumerGroupPositionResponse includes the list of partitions that this consumer is interested in. For each partition P1 that is assigned to this consumer AND whose position is interesting to some consumers of the consumer group, if the following following condition is satisfied , the consumer should execute the callback PartitionKeyRebalanceListener.onPartitionKeyMaybeRevoked(Collection<TopicPartition>) and include the position of the partition P1 in the ConsumerGroupPositionResponse.

Here is the condition for including partition P1. This condition optimizes the overhead of onPartitionKeyMaybeRevoked() callback such that consumer will not trigger callback and will not include the position of the partition in the ConsumerGroupPositionResponse when it knows for sure that this position will not unblock other consumers from starting to consume some partitions.

```
(The position of partition P1 has not been included in ConsumerGroupPositionResponse since this consumer
starts) ||
(The position of partition P1 has increased since the last ConsumerGroupPositionResponse AND it equals the high
watermark of P1 in the last FetchReponse) ||
(The largest leaderEpoch of those messages before position P1 has increased since the last
ConsumerGroupPositionResponse)
```

The motivation of this change is to allow consumers of the same consumer group to communicate the position of partitions with each other. This allows a consumer to start consuming beyond a given offset of a partition only after a certain condition is satisfied for the position of other partitions in the consumer group.

Note: the extra steps introduced in here is skipped for partitions of topics whose enable.ordered.delivery is set to false.

8) Changes in how consumer consumes partition

1. Consumer receives SyncGroupResponse, which contains its assigned partitions

2. Consumer gets the startPosition, i.e.the committedOffset, for its assigned partitions.

3. Consumer sends ListOffsetRequest to get the earliest offset for its assigned partitions.

4. For each partition P1 whose startPosition is not available, or whose startPosition equals the earliest offset, if the partition index >= initial_partition_count of the topic, consumer does the following before consuming the partition P1:

4.1 Consumer sends PartitionLeaderEpochsForPartitionsRequest to the coordinator to get the leaderEpochAfterCreation map for the partition P1, which can be read by broker from the corresponding partition znode.

4.2 There will be exactly one parition P3 that blocks this consumer from consuming the partition P1. Since producer uses linear hashing, partition P3 can be determined based on the set of partitions prior to partition expansion (i.e. keys of the leaderEpochAfterCreation map), the partition index of P1 and the initial_partition_count of the topic.

4.3 Then the consumer gets the oldLeaderEpoch for P3 from leaderEpochAfterCreation and sends OffsetsForLeaderEpochRequest to convert the oldLeaderEpoch to lowerOffsetThreshold, where lowerOffsetThreshold should be the last offset of messages published under the oldLeaderEpoch for partition P3.

4.4 Consumer includes the partition P3 in the ConsumerGroupPositionRequest and gets the corresponding position of partition P3 of the consumer group in the ConsumerGroupPositionResponse. If the position of P3 > lowerOffsetThreshold of the priorParition, then the consumer executes the callback PartitionKeyRebalanceListener.onPartitionKeyAssigned(Collection<TopicPartition> partitions) for P1 and starts to consume partition P1.

5. For each parrtition P1 assigned to this consumer, consumer queries the metadata to see if there any partition of this topic has been marked for deletion. If so, consumer does the following before deliverying message with offset T from this partition P1:

5.1 For every partition P3 that has been marked for deletion but not yet removed from this topic, consumer can determine whether partition P1 should be blocked on consuming all messages from P3. There will be at most O(logn) such partitions, where n is the maximum index of any partition of this topic. Since producer uses linear hashing, this can be determined based on index of partition P3, index of partition P1 and the initial_partition_count of the topic.

5.2 For every partition P3 that may block the consumption of partition P1, consumer sends PartitionLeaderEpochsForPartitionsRequest to coordinator to get the leaderEpochBeforeDeletion map for P3. Note that a partition is marked for deletion if and only if the leaderEpochBeforeDeletion map in its partition znode is not empty. Then the consumer gets the oldLeaderEpoch for P1 from the leaderEpochBeforeDeletion map of P3. Next, consumer sends OffsetsForLeaderEpochRequest to convert the oldLeaderEpoch to upperOffsetThreshold for P1. This results in a map from P3 -> upperOffsetThreshold for every partition P3 which blocks the consumption of partition P1.

5.2 For every partition P3 which may block the consumption of partition P1, consumer includes this partition in the ConsumerGroupPositionRequest and gets the corresponding position of this partition of the consumer group in the ConsumerGroupPositionResponse. Consumer also sends ListOffsetRequest to get the LogEndOffset for this partition.

5.3 For every partition P3 which may block the consumption of partition P1, if either the position of P3 of the consumer group has reached the LogEndOffset of P3 or the offset T <= upperOffsetThreshold, then the consumer will execute the callback PartitionKeyRebalanceListener. onPartitionKeyAssigned(Collection<TopicPartition> partitions) for P1 and start to deliver messages with offset >= T from partition P1. Otherwise, the consumer can not delievery message with offset >= T from partition P1.

6. Additionally, consumer should include the undeleted_partitiont_count in the FetchRequest. If the undeleted partition count is different from what broker expects for the given topic, broker will specify InvalidPartitionMetadataException for the given partition in the FetchRequest. This should trigger metadata update in consumer and cause consumer to rejoin the group. The purpose of this change is that, when we shrink partition of a given topic, consumers in the given consumer group can perform step 5 above to ensure that messages are delivered in order.

This change, combined with the controller change 1) and 2) described above, ensures in-order message delivery for keyed messages in the presence of partition creation and deletion.

Note: the extra steps introduced in here is skipped for partitions of topics whose enable.ordered.delivery is set to false.

9) Changes in how producer produces keyed messages to log compacted topics

If enable.ordered.delivery is true, message has non-null key, the topic is log compacted, and the initial_partition_count < undeleted_partition_count, producer may additionally send a message as described below:

- Say the original message to be sent by user has key K1 and value V1. Suppose this message is mapped to partition P1 by linear hashing where P1 >= initial_partition_count. And P2 is the partition which was split into {P1, P2} after partition expansion.

- Producer sends PartitionLeaderEpochsForPartitionsRequest to the coordinator to get the leaderEpochAfterCreation map for the partition P1.

- Producer gets the oldLeaderEpoch for partition P2 from the leaderEpochAfterCreation map. Then it sends OffsetsForLeaderEpochRequest to convert the oldLeaderEpoch to lowerOffsetThreshold, where lowerOffsetThreshold should be the last offset of messages published under the oldLeaderEpoch for partition P2.

- Producer sends ListOffsetRequest to get the earliest offset of P2. If the earliest offset > lowerOffsetThreshold of P2, producer sends message with key=K1 and value=null to P2 with ack=0 in adddition to sending the message key=K1 and value=V1 to P1.

The motivation of this change is to allow messages in the log compacted topic to be removed after a new message with the same key is produced after the partiton expansion.

Compatibility, Deprecation, and Migration Plan

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.

Rejected Alternatives

Future work

- Allow partition expansion when there exists partiton that has been marked for deletion but can not be deleted yet due to remaining data in the partition. This can likely be supported on top of this KIP by allowing an array of both leaderEpochAfterCreation and leaderEpochBeforeDeletion in the partition znode.