KIP-630: Kafka Raft Snapshot

Status

Current state: Adopted

Discussion thread: here

Vote thread: here

JIRA: ⚠ Unable to render Jira issues macro, execution error.

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

Terminology

1. **Kafka Controller:** The component that generates snapshots, reads snapshots and reads logs for voter replicas of the topic partition `__cluster_metadata`.
2. **Active Controller:** The Kafka Controller that is also a leader of the `__cluster_metadata` topic partition. This component is allowed to append to the log.
3. **Metadata Cache:** The component that generates snapshots, reads snapshots and reads logs for observer replicas of the topic partition `__cluster_metadata`. This is needed to reply to `Metadata` RPCs, for connection information to all of the brokers, etc.
4. **SnapshotId**: Each snapshot is uniquely identified by a `SnapshotId`, the epoch and end offset of the records in the replicated log included in the snapshot.

Motivation

In KIP-500 the Active Controller, which is the quorum leader from KIP-595, will materialize the entries in the metadata log into memory. Each broker in KIP-500, which will be a voter or an observer replica of the metadata log, will materialize the entries in the log into a metadata cache. We want to provide stronger guarantee on the consistency of the metadata than is possible today. If all entries in the metadata log are indefinitely retained, then the design described in KIP-595 achieves a consistent log and as a result a consistent metadata cache. If Kafka doesn't limit the size of this replicated log it can affect Kafka availability by:

1. Running out of disk space
2. Increasing the time needed to load the replicated log into memory
3. Increasing the time it takes for new brokers to catch up to the leader's log and hence join the cluster.

Kafka uses the cleanup policies `compact` and `delete` to limit the size of the log. The `delete` cleanup policy deletes data for a prefix of the log based on time. The `delete` cleanup policy is not viable when used on metadata log because the Kafka Controller and Metadata Cache will not be able to generate a consistent view from such a log.

The `compact` cleanup policy incrementally removes key/value records that are not needed to reconstruct the associated key/value state. During compaction the log cleaner keeps track of an `OffsetMap` which is a mapping from `key` to the latest offset in the section of the log being cleaned. The log cleaner assumes that any record with an offset less that the value in the `OffsetMap` given by the record's key is safe to remove. This solution will eventually result in one record for every key in log. This includes a tombstone record for key that have been deleted. Kafka removes this tombstone records after a configurable time. This can result in different logs in each replica which can result in different in-memory state after applying the log. We explore changes to the `compact` cleanup policy in the "Rejected Alternatives" section.

Type of State Machines and Operations

The type of in-memory state machines that we plan to implement for the Kafka Controller and metadata cache (see KIP-631) doesn't map very well to a key and offset based clean up policy. The data model for the Kafka Controller and metadata cache requires us to supports deltas/events. Currently the controller uses ZK for both storing the current state of the cluster and the deltas that it needs to apply. One example of this is topic deletion, when the user asks for a topic to be deleted 1) the Kafka Controller (`AdminManager` in the code) persists and commits the delete event, 2) applies the event to its in-memory state and sends RPC to all of the affected brokers, 3) persists and commits the changes to ZK. We don't believe that a key-value and offset based compaction model can be used for events/deltas in a way that it is intuitive to program against.

Another operation that the Kafka Controller models as an event is the `FenceBroker` message from KIP-631. Every time a broker is fenced because of controlled shutdown or failure to send a heartbeat it affects every topic partition assigned to the fenced broker. For details on the impact on replication of event operations (`FenceBroker`) vs key-value operations (`IsrChange`) see the section "Amount of Data Replicated".

Loading State and Frequency of Compaction

When starting a broker either because it is a new broker or it is restarting, loading the state represented by the `__cluster_metadata` and `__consumer_offsets` topic partition is required before the broker is available. By taking snapshots of the in-memory state as described below, Kafka can be much more efficient and aggressive on how often it generates the snapshot. By having snapshots in a separate file and not updating the replicated log the snapshot can include up to the high watermark.

For topics like __cluster_metadata and __consumer_offsets we can assume that the set of keys rarely changes but that values change frequently. With this assumption, to generate a snapshot from the in-memory state as propose in this document the broker needs to write O(the size of the data). For log compaction the broker has to write O(the size of the data) + O(the size of tombstones).

The difference is in the read pattern for the two solutions. For in-memory snapshots every log record will be read once to update the in-memory state. The Kafka Controller and the Group Coordinator will read from the head of the log. No additional reads are required to generate a snapshot.

For log compaction there are three reads:

1. One read to update the in-memory state.
2. One read to generate the map of keys to offsets.
3. One read to copy the live records from current log segments to the new log segments. These may read old offsets and may not be in the OS's file cache

If we assume an in-memory state of 100MB and a rate of change of 1MB/s, then

1. With in-memory snapshot the broker needs to read 1MB/s from the head of the log.
2. With log compaction the broker needs to
    a. read 1MB/s from the head of the log to update the in-memory state
    b. read 1MB/s to update the map of keys to offsets
    c. read 3MB/s (100MB from the already compacted log, 50MB from the new key-value records) from the older segments. The log will accumulate 50MB in 50 seconds worth of changes before compacting because the default configuration has a minimum clean ratio of 50%.

We mentioned `__consumer_offsets` in this section because in the future we are interested in using this mechanism for high-throughput topic partitions like the Group Coordinator and Transaction Coordinator.

Amount of Data Replicated

Because the Kafka Controller and Metadata Cache as described in KIP-631 will use events/deltas to represent some of its in-memory state changes the amount of data that needs to be written to the log and replicated to all of the brokers can be reduced. As an example if a broker goes offline and we assume that we have 100K topics each with 10 partitions with a replication factor of 3 and 100 brokers. In a well balanced cluster we can expect that no broker will have 2 or more partitions for the same topic. In this case the best we can do for `IsrChange` message and `FenceBroker` message are the following message sizes:

1. `IsrChange` has a size of 40 bytes. 4 bytes for partition id, 16 bytes for topic id, 8 bytes for ISR ids, 4 bytes for leader id, 4 bytes for leader epoch, 2 bytes for api key and 2 bytes for api version.
2. `FenceBroker` has a size of 8 bytes. 4 bytes for broker id, 2 bytes for api key and 2 bytes for api version.

When a broker is shutdown and the Active Controller uses `IsrChange` to include this information in the replicated log for the `__cluster_metadata` topic partition then the Kafka Controllers and each broker needs to persist 40 bytes per topic partitions hosted by the broker. Each broker will be a replica for 30,000 partitions, that is 1.14 MB that needs to get written in the replicated log by each broker. The Active Controller will need to replicate 1.14 MB to each broker in the cluster (99 brokers) or 113.29 MB.

When using events or deltas (Fence`Broker`) each broker needs to persist 8 bytes in the replicated log and 1.14 MB in the snapshot. The Active Controller will need to replica 8 bytes to each broker in the cluster (99 brokers) or .77 KB. Note that each broker still needs to persist 1.14 MB - the difference is that those 1.14 MB are not sent through the network.

Consistent Log and Tombstones

It is very important that the log remain consistent across all of the replicas after log compaction. The current implementation of log compaction in Kafka can lead to divergent logs because it is possible for slow replicas to miss the tombstone records. Tombstone records are deleted after a timeout and are not guarantee to have been replicated to all of the replicas before they are deleted. It should be possible to extend log compaction and the fetch protocol as explained in the Rejected Alternatives "Just fix the tombstone GC issue with log compaction". We believed that this conservative approach is equally, if not more complicated than snapshots as explained in this proposal and more importantly it doesn't address the other items in the motivation section above.

Overview

This KIP assumes that KIP-595 has been approved and implemented. With this improvement replicas will continue to replicate their log among each others. This means that voters (leader and followers) and observers will have the replicated log on disk. Replicas will snapshot their log independent of each other. Snapshots will be consistent because the logs are consistent across replicas. Follower and observer replicas fetch the snapshots from the leader when they attempt to fetch an offset from the leader and the leader doesn't have that offset in the log.

Generating and loading the snapshot will be delegated to the Kafka Controllers and Metadata Cache. The Kafka Controllers and Metadata Cache will notify the Kafka Raft layer when it has generated a snapshot and the end offset of the snapshot. The Kafka Raft layer will notify the Kafka Controller or Metadata Cache when a new snapshot has been fetched from the Active Controller. See section "Interaction with the Kafka Controller or Metadata Cache" and KIP 595 for more details.

This KIP focuses on the changes required for use by the `__cluster_metadata` topic partition but we should be able to extend it to use it in other internal topic partitions like `__consumer_offsets` and `__transaction`.

Proposed Changes

The `__cluster_metadata` topic will have `snapshot` as the `cleanup.policy`. The Kafka Controller and Metadata Cache will have an in-memory representation of the replicated log up to at most the high-watermark. When performing a snapshot the Kafka Controller and Metadata Cache will serialize this in-memory state to disk. This snapshot file on disk is described by the end offset and epoch from the replicated log that has been included.

The Kafka Controller and Metadata Cache will notify the Kafka Raft client when it has finished generating a new snapshots. It is safe to truncate a prefix of the log up to the latest snapshot. The `__cluster_metadata` topic partition will have the latest snapshot and zero or more older snapshots. These additional snapshots would have to be deleted, this is described in "When to Delete Snapshots".

Design

Visually a Kafka Raft topic partition would look as follow:

```
Kafka Replicated Log:

LogStartOffset  --             high-watermark --           LEO --
                 V                             V               V
               -----------------------------------------------
       offset: | x | ... | y - 1 | y |  ...  |   |  ...  |   |
       epoch:  | b | ... | c     | d |  ...  |   |  ...  |   |
               -----------------------------------------------

Kafka Snapshot Files:

<topic_name>-<partition_index>/x-a.checkpoint
<topic_name>-<partition_index>/y-c.checkpoint
```

Note: The extension `checkpoint` will be used since Kafka already has a file with the `snapshot` extension.

Legend

1. LEO - log end offset - the next offset to be written to disk.
2. high-watermark - the largest offset and epoch that has been replicated to N/2 + 1 replicas.
3. LogStartOffset - log start offset - the smallest offset in the replicated log.

In the example above, offset=x - 1, epoch=a does not appear in the replicated log because it is before the `LogStartOffset` (x, b). If the Kafka topic partition leader receives a fetch request with an offset and last fetched epoch greater than or equal to the `LogStartOffset` then the leader handles the request as describe in KIP-595. Otherwise, the leader will respond with the `SnapshotId,` epoch and end offset, of the latest snapshot `(y, c)`.

Adding snapshots to KIP-595 changes how voters grant votes and how candidate request votes. See section "Changes to Leader Election" for details on those changes.

Snapshot Format

The Kafka Controller and Metadata Cache are responsible for the content of the snapshot. Each snapshot is uniquely identified by a `SnapshotId`, the epoch and end offset of the records in the replicated log included in the snapshot. The snapshot will be stored in the topic partition directory and will have a name of `<SnapshotId.EndOffset>-<SnapshotId.Epoch>.checkpoint`. For example, for the topic __cluster_metadata, partition 0, snapshot end offset 5120793 and snapshot epoch 2, the full filename will be `__cluster_metadata-0/00000000000005120793-00000000000000000002.checkpoint`.

The snapshot epoch will be used when ordering snapshots and more importantly when setting the `LastFetchedEpoch` field in the Fetch request. It is possible for a follower to have a snapshot and an empty log. In this case the follower will use the epoch of the snapshot when setting the `LastFetchEpoch` in the Fetch request.

The disk format for snapshot files will be the same as the version 2 of the log format. This is the version 2 log format for reference:

```
RecordBatch => BatchHeader [Record]

BatchHeader
   BaseOffset => Int64
   Length => Int32
   PartitionLeaderEpoch => Int32
   Magic => Int8
   CRC => Uint32
   Attributes => Int16
   LastOffsetDelta => Int32 // also serves as LastSequenceDelta
   FirstTimestamp => Int64
   MaxTimestamp => Int64
   ProducerId => Int64
   ProducerEpoch => Int16
   BaseSequence => Int32

Record =>
   Length => Varint
   Attributes => Int8
   TimestampDelta => Varlong
   OffsetDelta => Varint
   Key => Bytes
   Value => Bytes
   Headers => [HeaderKey HeaderValue]
     HeaderKey => String
     HeaderValue => Bytes
```

Using version 2 of the log format will allow the Kafka Controllers and Metadata Cache to compress records and identify corrupted records in the snapshot. Even though snapshot use the log format for storing this state there is no requirement:

1. To use valid `BaseOffset` and `OffsetDelta` in the `BatchHeader` and `Record` respectively.
2. For the records in the snapshot to match the records in the replicated log.

Snapshot Control Records

To allow the KRaft implementation to include additional information on the snapshot without affect the Kafka Controller and Metadata cache the snapshot will include two control record batches. The control record batch `SnapshotHeaderRecord` will always be the first record batch in a snapshot. The control record batch `SnapshotFooterRecord` will be the last record batch in a snapshot. These two records will have the following schema.

Snapshot Header Schema

```
{
  "type": "data",
  "name": "SnapshotHeaderRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    {"name": "Version", "type": "int16", "versions": "0+",
      "about": "The version of the snapshot header record"},
    { "name": "LastContainedLogTimestamp", "type": "int64", "versions": "0+",
      "about": "The append time of the last record from the log contained in this snapshot" }
  ]
}
```

Snapshot Footer Schema

```
{
  "type": "data",
  "name": "SnapshotFooterRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Version", "type": "int16", "versions": "0+",
      "about": "The version of the snapshot footer record" }
  ]
}
```

When to Snapshot

If the Kafka Controller or Metadata Cache generates a snapshot too frequently then it can negatively affect the performance of the disk. If it doesn't generate a snapshot often enough then it can increase the amount of time it takes to load its state into memory and it can increase the amount space taken up by the replicated log. The Kafka Controller and Metadata Cache will have new configuration options `metadata.snapshot.min.changed_records.ratio` and `metadata.log.max.record.bytes.between.snapshots`. Both of these configuration need to be satisfied before the Controller or Metadata Cache will generate a new snapshot.

1. `metadata.snapshot.min.changed_records.ratio` is the minimum number of snapshot records that have changed (deleted or modified) between the latest snapshot and the current in-memory state. Note that new snapshot records don't count against this ratio. If a new snapshot record was added since that last snapshot then it doesn't affect the dirty ratio. If a snapshot record was added and then modify or deleted then it counts against the ratio.
2. `metadata.log.max.record.bytes.between.snapshots` is the minimum size of the records in the replicated log between the latest snapshot and then high-watermark.

When to Increase the LogStartOffset

To minimize the need to send snapshots to slow replicas, the leader can delay increasing the log start offset up to the minimum offset that it knows has been replicate to all of the live replicas. The leader's `LogStartOffset` can be increased to an offset `X` if the following is true:

1. There is a snapshot for the topic partition with an end offset of `X`.
2. One of the following is true:
    a. All of the live replicas (followers and observers) have replicated `LogStartOffset` or
    b. `LogStartOffset` is `metadata.start.offset.lag.time.max.ms` milliseconds old.

Followers and observers will increase their log start offset to the value sent on the fetch response as long as the local Kafka Controller and Metadata Cache has generated a snapshot with an end offset greater than or equal to the new log start offset. In other words for followers and observers the local log start offset is the minimum of the leader's log start offset and the largest local snapshot.

Kafka allows the clients to delete records that are less than a given offset by using the `DeleteRecords` RPC . Those requests will be rejected for the __c`luster_metadata` topic with a POLICY_VIOLATION error.

When to Delete Snapshots

The broker will delete any snapshot with a end offset and epoch (`SnapshotId`) less than the `LogStartOffset`. That means that the question of when to delete a snapshot is the same as when to increase the `LogStartOffset` which is covered in the section “When To Increase the LogStartOffset”.

Interaction with the Kafka Controller and Metadata Cache

This section describe the interaction between the state machine and Kafka Raft.

From Kafka Raft to the Kafka Controller and Metadata Cache

1. Notify when a snapshot is available because of `FetchSnapshot`.

From the Kafka Controller and Metadata Cache to Kafka Raft

1. Notify when snapshot finished. This includes the end offset and epoch.

Loading Snapshots

There are two cases when the Kafka Controller or Metadata Cache needs to load a snapshot:

1. When it is booting.
2. When the follower and observer replicas finishes fetching a new snapshot from the leader.

After loading the snapshot, the Kafka Controller and Metadata Cache can apply operations in the replicated log with an offset greater than or equal the snapshot's end offset.

Changes to Leader Election

The followers of the `__cluster_metadata` topic partition need to take the snapshots into account when granting votes and asking for votes (becoming a candidate). Conceptually, a follower implements the following algorithm:

1. Follower sends a fetch request
2. Leader replies with a snapshot epoch and end offset.
3. Follower pauses fetch requests
4. Follower fetches snapshot chunks from the leader into a temporary snapshot file (`<EndOffset>-<Epoch>.checkpoint.part`)
5. Validate fetched snapshot
    a. All snapshot chunks are fetched
    b. Verify the CRC of the records in the snapshot
    c. Atomically move the temporary snapshot file (`<EndOffset>-<Epoch>.checkpoint.part`) to the permanent location (`<EndOffset>-<Epoch>.checkpoint`)
6. Follower resumes fetch requests by
    a. Setting the `LogStartOffset` to the snapshot's end offset.
    b. Setting the `LEO` or `FetchOffset` in the fetch request to the snapshot's end offset.
    c. Setting the `LastFetchedEpoch` in the fetch request to the snapshot's epoch.

Assuming that the snapshot fetched in bullet 4 has an epoch `E1` and a offset `O1`. The state of the follower between bullet 4. and 5. is as follow:

1. Contains a snapshot with epoch `E1` and end offset `O1`.
2. Contains zero or more snapshots older/less than epoch `E1` and end offset `O1`.
3. Contains a replicated log with `LEO` older/less than epoch `E1` and end offset `O1`.

In this case the follower needs to use the snapshot with epoch `E1` and end offset `O1`, and not the replicated log when granting votes and requesting votes. In the general case, the latest epoch and end offset for a replica is the latest snapshot or replicated log.

Validation of Snapshot and Log

For each replica we have the following invariant:

1. If the `LogStartOffset` is 0, there are zero or more snapshots where the end offset of the snapshot is between `LogStartOffset` and the `High-Watermark`.
2. If the `LogStartOffset` is greater than 0, there are one or more snapshots where the end offset of the snapshot is between `LogStartOffset` and the `High-Watermark`.

If the latest snapshot has an epoch `E1` and end offset `O1` and is it newer than the `LEO` of the replicated log, then the replica must set the `LogStartOffset` and `LEO` to `O1`.

Public Interfaces

Topic configuration

1. The `__cluster_metadata` topic will have a `cleanup.policy` value of `snapshot`. This configuration can only be read. Updates to this configuration will not be allowed.
2. `metadata.snapshot.min.changed_records.ratio` - The minimum ratio of snapshot records that have to change before generating a snapshot. See section "When to Snapshot". The default is .5 (50%).

3. `metadata.log.max.record.bytes.between.snapshots` - This is the minimum number of bytes in the replicated log between the latest snapshot and the high-watermark needed before generating a new snapshot. See section "When to Snapshot". The default is 20MB.
4. `metadata.start.offset.lag.time.max.ms` - The maximum amount of time that leader will wait for an offset to get replicated to all of the live replicas before advancing the `LogStartOffset`. See section “When to Increase the LogStartOffset”. The default is 7 days.

Network Protocol

Fetch

Request Schema

This KIP doesn’t require any changes to the fetch request outside of what is proposed in KIP-595.

Response Schema

The fetch response proposed in KIP-595 will be extended such that it includes new optional fields for snapshot end offset and snapshot epoch.

```
{
  "apiKey": 1,
  "type": "response",
  "name": "FetchResponse",
  "validVersions": "0-12",
  "flexibleVersions": "12+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "7+", "ignorable": false,
      "about": "The top level response error code." },
    { "name": "SessionId", "type": "int32", "versions": "7+", "default": "0", "ignorable": false,
      "about": "The fetch session ID, or 0 if this is not part of a fetch session." },
    { "name": "Topics", "type": "[]FetchableTopicResponse", "versions": "0+",
      "about": "The response topics.", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]FetchablePartitionResponse", "versions": "0+",
        "about": "The topic partitions.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no fetch error." },
        { "name": "HighWatermark", "type": "int64", "versions": "0+",
          "about": "The current high water mark." },
        { "name": "LastStableOffset", "type": "int64", "versions": "4+", "default": "-1", "ignorable": true,
          "about": "The last stable offset (or LSO) of the partition. This is the last offset such that the
state of all transactional records prior to this offset have been decided (ABORTED or COMMITTED)" },
        { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": true,
          "about": "The current log start offset." },
        { "name": "DivergingEpoch", "type": "EpochEndOffset", "versions": "12+", "taggedVersions": "12+",
"tag": 0,
          "about": "In case divergence is detected based on the `LastFetchedEpoch` and `FetchOffset` in the
request, this field indicates the largest epoch and its end offset such that subsequent records are known to
diverge",
          "fields": [
            { "name": "Epoch", "type": "int32", "versions": "12+", "default": "-1" },
            { "name": "EndOffset", "type": "int64", "versions": "12+", "default": "-1" }
        ]},
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
          "versions": "12+", "taggedVersions": "12+", "tag": 1, fields": [
          { "name": "LeaderId", "type": "int32", "versions": "0+",
            "about": "The ID of the current leader or -1 if the leader is unknown."},
          { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
            "about": "The latest known leader epoch"}
        ]},
        // ---------- Start of new field ----------
        { "name": "SnapshotId", "type": "SnapshotId",
          "versions": "12+", "taggedVersions": "12+", "tag": 2,
          "about": "In the case of fetching an offset less than the LogStartOffset, this is the end offset and
```

```
epoch that should be used in the FetchSnapshot request.",
          "fields": [
            { "name": "EndOffset", "type": "int64", "versions": "0+"},
            { "name": "Epoch", "type": "int32", "versions": "0+"}
        ]},
        // ---------- End new field ----------
        { "name": "Aborted", "type": "[]AbortedTransaction", "versions": "4+", "nullableVersions": "4+",
"ignorable": false,
          "about": "The aborted transactions.",  "fields": [
          { "name": "ProducerId", "type": "int64", "versions": "4+", "entityType": "producerId",
            "about": "The producer id associated with the aborted transaction." },
          { "name": "FirstOffset", "type": "int64", "versions": "4+",
            "about": "The first offset in the aborted transaction." }
        ]},
        { "name": "PreferredReadReplica", "type": "int32", "versions": "11+", "ignorable": true,
          "about": "The preferred read replica for the consumer to use on its next fetch request"},
        { "name": "Records", "type": "bytes", "versions": "0+", "nullableVersions": "0+",
          "about": "The record data." }
      ]}
    ]}
  ]
}
```

Handling Fetch Request

The leader's handling of fetch request will be extended such that if `FetchOffset` is less than `LogStartOffset` then the leader will respond with a `SnapshotId` of the latest snapshot.

Handling Fetch Response

The replica's handling of fetch response will be extended such that if `SnapshotId` is set then the follower will pause fetching of the log, and start fetching the snapshot from the leader. The value use in `SnapshotId` will be used in the `FetchSnapshot` requests. Once the snapshot has been fetched completely, the local log will be truncated and fetch will resume.

Fetching Snapshots

Request Schema

```
{
  "apiKey": 59,
  "type": "request",
  "name": "FetchSnapshotRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
"taggedVersions": "0+", "tag": 0,
      "about": "The clusterId if known, this is used to validate metadata fetches prior to broker registration"
},
    { "name": "ReplicaId", "type": "int32", "versions": "0+", "default": "-1", "entityType": "brokerId",
      "about": "The broker ID of the follower" },
    { "name": "MaxBytes", "type": "int32", "versions": "0+",
      "about": "The maximum bytes to fetch from all of the snapshots." },
    { "name": "Topics", "type": "[]TopicSnapshot", "versions": "0+",
      "about": "The topics to fetch.", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The name of the topic to fetch" },
      { "name": "Partitions", "type": "[]PartitionSnapshot", "versions": "0+",
        "about": "The partitions to fetch", "fields": [
        { "name": "Partition", "type": "int32", "versions": "0+",
          "about": "The partition index" },
        { "name": "CurrentLeaderEpoch", "type": "int32", "versions": "0+",
          "about": "The current leader epoch of the partition, -1 for unknown leader epoch" },
        { "name": "SnapshotId", "type": "SnapshotId", "versions": "0+",
          "about": "The snapshot endOffset and epoch to fetch",
          "fields": [
          { "name": "EndOffset", "type": "int64", "versions": "0+" },
          { "name": "Epoch", "type": "int32", "versions": "0+" }
        ]},
```

```
        { "name": "Position", "type": "int64", "versions": "0+",
          "about": "The byte position within the snapshot to start fetching from" }
      ]}
    ]}
  ]
}
```

Response Schema

```
{
  "apiKey": 59,
  "type": "response",
  "name": "FetchSnapshotResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+", "ignorable": false,
      "about": "The top level response error code." },
    { "name": "Topics", "type": "[]TopicSnapshot", "versions": "0+",
      "about": "The topics to fetch.", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The name of the topic to fetch." },
      { "name": "Partitions", "type": "[]PartitionSnapshot", "versions": "0+",
        "about": "The partitions to fetch.", "fields": [
        { "name": "Index", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no fetch error." },
        { "name": "SnapshotId", "type": "SnapshotId", "versions": "0+",
          "about": "The snapshot endOffset and epoch fetched",
          "fields": [
          { "name": "EndOffset", "type": "int64", "versions": "0+" },
          { "name": "Epoch", "type": "int32", "versions": "0+" }
        ]},
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
          "versions": "0+", "taggedVersions": "0+", "tag": 0, "fields": [
          { "name": "LeaderId", "type": "int32", "versions": "0+",
            "about": "The ID of the current leader or -1 if the leader is unknown."},
          { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
            "about": "The latest known leader epoch"}
        ]},
        { "name": "Size", "type": "int64", "versions": "0+",
          "about": "The total size of the snapshot." },
        { "name": "Position", "type": "int64", "versions": "0+",
          "about": "The starting byte position within the snapshot included in the Bytes field." },
        { "name": "UnalignedRecords", "type": "records", "versions": "0+",
          "about": "Snapshot data in records format which may not be aligned on an offset boundary" }
       ]}
    ]}
  ]
}
```

Request Handling

When the leader receives a `FetchSnapshot` request it would do the following:

1. Find the snapshot for `SnapshotId` for the topic `Name` and partition `Index`.
2. Set the `Size` of each snapshot.
3. Send the bytes in the snapshot from `Position`. If there are multiple partitions in the `FetchSnapshot` request, then the leader will evenly distribute the number of bytes sent across all of the partitions. The leader will not send more bytes in the response than `ResponseMaxBytes`, which is the minimum of `MaxBytes` in the request and the value configured in `replica.fetch.response.max.bytes`.
    a. Each topic partition is guaranteed to receive at least the average of `ResponseMaxBytes` if that snapshot has enough bytes remaining.
    b. If there are topic partitions with snapshots that have remaining bytes less than the average `ResponseMaxBytes`, then those bytes may be used to send snapshot bytes for other topic partitions.

Errors:

1. `SNAPSHOT_NOT_FOUND` - when the fetch snapshot request specifies a `SnapshotId` that doesn't exists on the leader.
2. `POSITION_OUT_OF_RANGE` - when the fetch snapshot request specifies a position that is greater than the size of the snapshot.
3. `NOT_LEADER_FOR_PARTITION` - when the fetch snapshot request is sent to a replica that is not the leader.

Response Handling

1. Copy the bytes in `UnalignedRecords` to the local snapshot file name `<SnapshotId.EndOffset>-<SnapshotId.Epoch>.checkpoint.part` starting at file position `Position`.
2. If `Size` is greater than the size of the current snapshot file, then the follower is expected to send another `FetchSnapshotRequest` with a `Position` set to the response's `Position` plus the response's `len(UnalignedRecords)`.
3. If `Size` is equal to the size of the current snapshot file, then fetching the snapshot has finished.

    a. Validate the snapshot file.
    b. Atomically move the file named `<SnapshotId.EndOffset>-<SnapshotId.Epoch>.checkpoint.part` to a file named `<SnapshotId.EndOffset>-<SnapshotId.Epoch>.checkpoint`.
    c. Notify the Kafka Controller or Metadata Cache that a new snapshot is available.
    d. The Kafka Controller and Metadata Cache will load this new snapshot and read records from the log after the snapshot's end offset.

Errors:

1. `SNAPSHOT_NOT_FOUND`, `POSITION_OUT_OF_RANGE`, `NOT_LEADER_FOR_PARTITION` - The follower can rediscover the new snapshot by sending another `Fetch` request.

Metrics

Full Name	Description
kafka.controller:type=KafkaController, name=GenSnapshotLatencyMs	A histogram of the amount of time it took to generate a snapshot.
kafka.controller:type=KafkaController, name=LoadSnapshotLatencyMs	A histogram of the amount of time it took to load the snapshot.
kafka.controller:type=KafkaController,name=SnapshotSizeBytes	Size of the latest snapshot in bytes.
kafka.controller:type=KafkaController,name=SnapshotLag	The number of offsets between the largest snapshot offset and the high-watermark.

Compatibility, Deprecation, and Migration Plan

This KIP is only implemented for the internal topic `__cluster_metadata`. The inter-broker protocol (`IBP`) will be increased to indicate that all of the brokers in the cluster support KIP-595 and KIP-630.

Rejected Alternatives

Append the snapshot to the Kafka log: Instead of having a separate file for snapshots and separate RPC for downloading the snapshot, the leader could append the snapshot to the log. This design has a few drawbacks. Because the snapshot gets appended to the head of the log that means that it will be replicated to every replica in the partition. This will cause Kafka to use more network bandwidth than necessary.

Use Kafka Log infrastructure to store snapshots: Kafka Log could be used to store the snapshot to disk and the Fetch request on a special topic partition could be used to the replicate the snapshot to the replicas. There are a couple observations with this design decision:

1. The `Fetch` RPC is typically used to download a never ending stream of records. Snapshot have a known size and the fetching of a snapshot ends when all of those bytes are received by the replica. The `Fetch` RPC includes a few fields for dealing with this access pattern that are not needed for fetching snapshots.
2. This solution ties the snapshot format to Kafka's log record format. To lower the overhead of writing and loading the snapshot by the Kafka Controller and Metadata Cache, it is important to use a format that is compatible with the in-memory representation of the Kafka Controller and Metadata Cache.

Change leader when generating snapshots: Instead of supporting concurrent snapshot generation and writing to the in-memory state, we could require that only non-leaders generate snapshots. If the leader wants to generate a snapshot then it would relinquish leadership. We decided to not implement this because:

1. All replicas/brokers in the cluster will have a metadata cache which will apply the replicated log and allow read-only access by other components in the Kafka Broker.
2. Even though leader election within the replicas in the replicated log may be efficient, communicating this new leader to all of the brokers and all of the external clients (admin clients) may have a large latency.

3. Kafka wants to support quorums where there is only one voter replica for the Kafka Controller.

Just fix the tombstone GC issue with log compaction: We have also discussed about fixing the tombstone garbage collecting issue as listed in motivation instead of introducing the snapshot mechanism. The key idea is that fetchers either from follower or from consumer needs to be notified if they "may" be missing some tombstones that have deprecated some old records. Since the offset span between the deprecated record and the tombstone for the same key can be arbitrarily long — for example, we have a record with key A at offset 0, and then a tombstone record for key A appended a week later, at offset 1 billion — we'd have to take a conservative approach here. More specifically:

1. The leader would maintain an additional offset marker up to which we have deleted tombstones, i.e. all tombstones larger than this offset are still in the log. Let's call it a **tombstone horizon marker**. This horizon marker would be returned in each fetch response.
2. Follower replicas would not do delete its tombstones beyond the received leader's horizon marker, and would maintain it's own marker as well which should always be smaller than leader's horizon marker offset.
3. All fetchers (follower and consumer) would keep track of the horizon marker from the replica it is fetching from. The expectation is that leaders should only delete tombstones in a very lazy manner, controlled by the existing config, so that the fetching position should be larger than the horizon marker most of the time. However, if the fetcher does found its fetching position below the horizon marker, then it needs to handle it accordingly:
    a. For consumer fetchers, we need to throw an extended error of `InvalidOffsetException` to notify the users that we may missed some tombstones, and let the users to reset is consumed record states — for example, if the caller of consumer is a streams client, and it is building a local state from the fetched records, that state need to be wiped.
    b. For replica fetchers, we need to wipe the local log all together and then restart fetching from the starting offset.
    c. For either types of fetchers, we will set an internal flag, let's call it **unsafe-bootstrapping** mode to true, while remembering the horizon marker offset at the moment from the replica it is fetching from.
        i. When this flag is set to true, we would not fall back to a loop of this handling logic even when the fetching position is smaller than the horizon marker, since otherwise we would fall into a loop of rewinding.
        ii. However, during this period of time, if the horizon marker offsets get updated from the fetch response, meaning that some tombstones are deleted again, then we'd need to retry this unsafe-bootstrap from step one.
        iii. After the fetching position as advanced beyond the remembered horizon marker offset, we would reset this flag to false to go back to the normal mode.

We feel this conservative approach is equally, if not more complicated than adding a snapshot mechanism since it requires rather tricky handling logic on the fetcher side; plus it only resolves one of the motivations — the tombstone garbage collection issue.

Another thought against this idea is that for now most of the internal topics (controller, transaction coordinator, group coordinator) that would need the snapshot mechanism describe in this KIP can hold their materialized cache in memory completely, and hence generating snapshots should be comparably more efficient to execute.