KIP-405: Kafka Tiered Storage

Authors Satish Duggana, Sriharsha Chintalapani, Ying Zheng, Suresh Srinivas

Status

Current State: "*Accepted*"

Discussion Thread: here

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

Motivation

Kafka is an important part of data infrastructure and is seeing significant adoption and growth. As the Kafka cluster size grows and more data is stored in Kafka for a longer duration, several issues related to scalability, efficiency, and operations become important to address.

Kafka stores the messages in append-only log segments on local disks on Kafka brokers. The retention period for the log is based on `log.retention` that can be set system-wide or per topic. Retention gives the guarantee to consumers that even if their application failed or was down for maintenance, it can come back within the retention period to read from where it left off without losing any data.

The total storage required on a cluster is proportional to the number of topics/partitions, the rate of messages, and most importantly the retention period. A Kafka broker typically has a large number of disks with a total storage capacity of 10s of TBs. The amount of data locally stored on a Kafka broker presents many operational challenges.

Kafka as a long-term storage service

Kafka has grown in adoption to become the entry point of all of the data. It allows users to not only consume data in real-time but also gives the flexibility to fetch older data based on retention policies. Given the simplicity of Kafka protocol and wide adoption of consumer API, allowing users to store and fetch data with longer retention help make Kafka one true source of data.

Currently, Kafka is configured with a shorter retention period in days (typically 3 days) and data older than the retention period is copied using data pipelines to a more scalable external storage for long-term use, such as HDFS. This results in data consumers having to build different versions of applications to consume the data from different systems depending on the age of the data.

Kafka cluster storage is typically scaled by adding more broker nodes to the cluster. But this also adds needless memory and CPUs to the cluster making overall storage cost less efficient compared to storing the older data in external storage. A larger cluster with more nodes also adds to the complexity of deployment and increases the operational costs.

Kafka local storage and operational complexity

When a broker fails, the failed node is replaced by a new node. The new node must copy all the data that was on the failed broker from other replicas. Similarly, when a new Kafka node is added to scale the cluster storage, cluster rebalancing assigns partitions to the new node which also requires copying a lot of data. The time for recovery and rebalancing is proportional to the amount of data stored locally on a Kafka broker. In setups that have many Kafka clusters running 100s of brokers, a node failure is a common occurrence, with a lot of time spent in recovery making operations difficult and time-consuming.

Reducing the amount of data stored on each broker can reduce the recovery/rebalancing time. But it would also necessitate reducing the log retention period impacting the time available for application maintenance and failure recovery.

Kafka in cloud

On-premise Kafka deployments use hardware SKUs with multiple high capacity disks to maximize the i/o throughput and to store the data for the retention period. Equivalent SKUs with similar local storage options are either unavailable or they are very expensive in the cloud. There are more available options for SKUs with lesser local storage capacity as Kafka broker nodes and they are more suitable in the cloud.

Solution - Tiered storage for Kafka

Kafka data is mostly consumed in a streaming fashion using tail reads. Tail reads leverage OS's page cache to serve the data instead of disk reads. Older data is typically read from the disk for backfill or failure recovery purposes and is infrequent.

In the tiered storage approach, Kafka cluster is configured with two tiers of storage - local and remote. The local tier is the same as the current Kafka that uses the local disks on the Kafka brokers to store the log segments. The new remote tier uses systems, such as HDFS or S3 to store the completed log segments. Two separate retention periods are defined corresponding to each of the tiers. With remote tier enabled, the retention period for the local tier can be significantly reduced from days to few hours. The retention period for remote tier can be much longer, days, or even months. When a log segment is rolled on the local tier, it is copied to the remote tier along with the corresponding indexes. Latency sensitive applications perform tail reads and are served from local tier leveraging the existing Kafka mechanism of efficiently using page cache to serve the data. Backfill and other applications recovering from a failure that needs data older than what is in the local tier are served from the remote tier.

This solution allows scaling storage independent of memory and CPUs in a Kafka cluster enabling Kafka to be a long-term storage solution. This also reduces the amount of data stored locally on Kafka brokers and hence the amount of data that needs to be copied during recovery and rebalancing. Log segments that are available in the remote tier need not be restored on the broker or restored lazily and are served from the remote tier. With this, increasing the retention period no longer requires scaling the Kafka cluster storage and the addition of new nodes. At the same time, the overall data retention can still be much longer eliminating the need for separate data pipelines to copy the data from Kafka to external stores, as done currently in many deployments.

Goals

Extend Kafka's storage beyond the local storage available on the Kafka cluster by retaining the older data in an external store, such as HDFS or S3 with minimal impact on the internals of Kafka. Kafka behavior and operational complexity must not change for existing users that do not have tiered storage feature configured.

Non-Goals

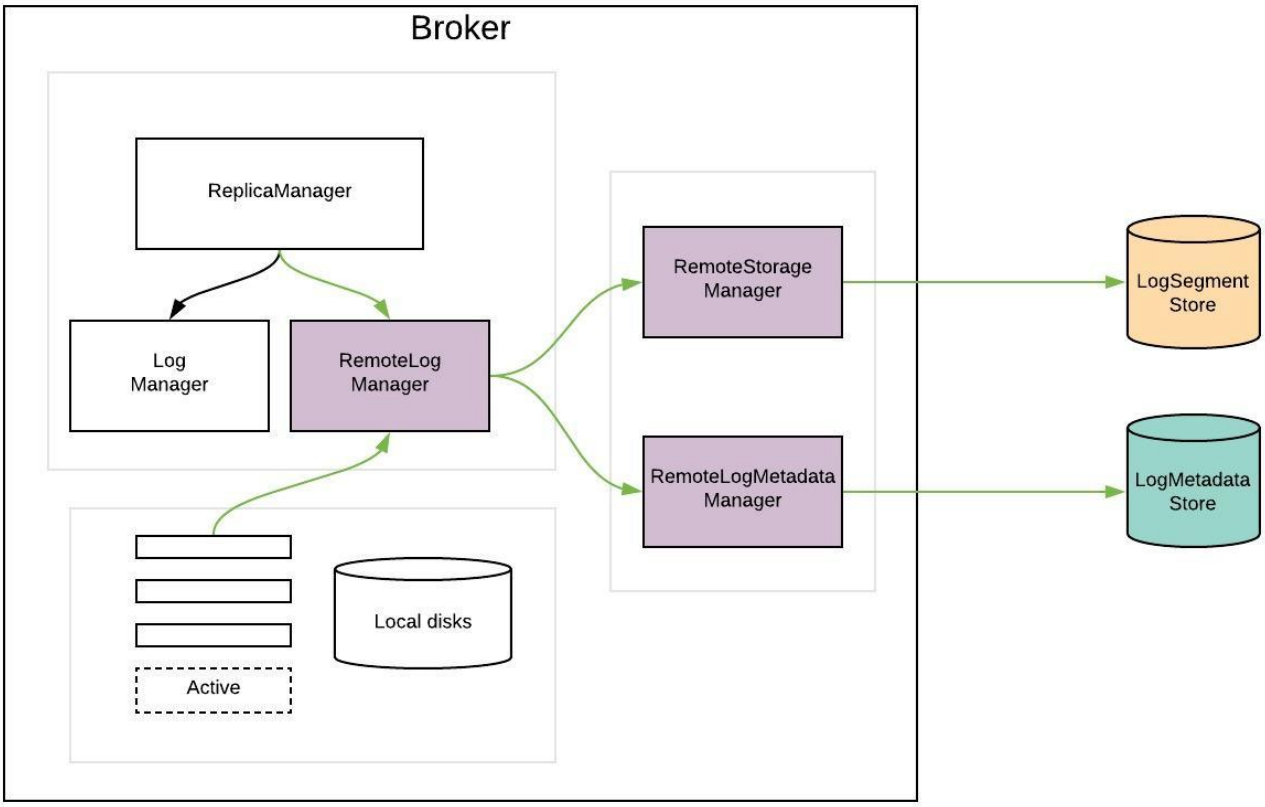
Tiered storage does not replace ETL pipelines and jobs. Existing ETL pipelines continue to consume data from Kafka as is, albeit with data in Kafka having a much longer retention period.

It does not support compact topics with tiered storage. Topic created with the effective value for remote.storage.enable as true, can not change its retention policy from delete to compact.

It does not support JBOD feature with tiered storage.

Proposed Changes

High-level design



RemoteLogManager (RLM) is a new component which

- receives callback events for leadership changes and stop/delete events of topic partitions on a broker.
- delegates copy, read, and delete of topic partition segments to a pluggable storage manager(viz *RemoteStorageManager*) implementation and maintains respective remote log segment metadata through *RemoteLogMetadataManager*.

`*RemoteLogManager*` is an internal component and it is not a public API.

`RemoteStorageManager` is an interface to provide the lifecycle of remote log segments and indexes. More details about how we arrived at this interface are discussed in the document. We will provide a simple implementation of RSM to get a better understanding of the APIs. HDFS and S3 implementation are planned to be hosted in external repos and these will not be part of Apache Kafka repo. This is inline with the approach taken for Kafka connectors.

`RemoteLogMetadataManager` is an interface to provide the lifecycle of metadata about remote log segments with strongly consistent semantics. There is a default implementation that uses an internal topic. Users can plugin their own implementation if they intend to use another system to store remote log segment metadata.

RemoteLogManager (RLM)

RLM creates tasks for each leader or follower topic partition, which are explained in detail here.

- RLM Leader Task
  - It checks for rolled over LogSegments (which have the last message offset less than last stable offset of that topic partition) and copies them along with their offset/time/transaction/producer-snapshot indexes and leader epoch cache to the remote tier. It also serves the fetch requests for older data from the remote tier. Local logs are not cleaned up till those segments are copied successfully to remote even though their retention time/size is reached.
- RLM Follower Task
  - It keeps track of the segments and index files on the remote tier by looking into *RemoteLogMetdataManager*. RLM follower can also serve reading old data from the remote tier.

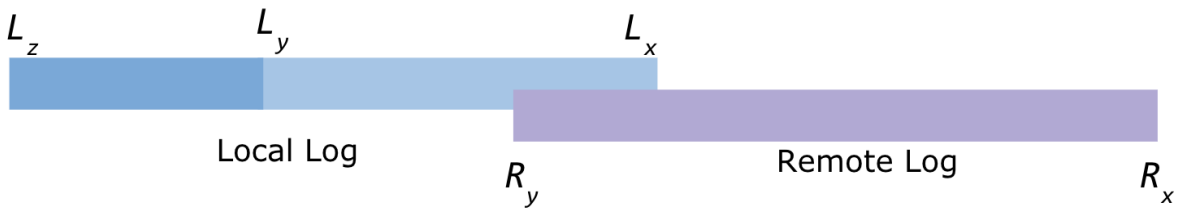
RLM maintains a bounded cache(possibly LRU) of the index files of remote log segments to avoid multiple index fetches from the remote storage. They are stored in a directory `remote-log-index-cache` under log dir. These indexes can be used in the same way as local segment indexes are used. User can configure `remote.log.index.file.cache.total.size.mb` to set the total size that can be used for these index files.

The earlier approach consists of pulling the remote log segment metadata from remote log storage APIs as mentioned in the earlier RemoteStorageManager_Old section. This approach worked fine for storages like HDFS. One of the problems of relying on remote storage to maintain metadata is that tiered-storage needs to be strongly consistent, with an impact not only on the metadata itself (e.g. LIST in S3) but also on the segment data (e.g. GET after a DELETE in S3). Also, the cost (and to a lesser extent performance) of maintaining metadata in remote storage needs to be factored in. In the case of S3, frequent LIST APIs incur huge costs.

So, remote storage is separated from the remote log metadata store and introduced *RemoteStorageManager* and *RemoteLogMetadataManager respectively*. You can see the discussion details in the doc located here.

Local and Remote log offset constraints

Below are the leader topic partition's log offsets



Lx = Local log start offset L_z *= Local log end offset* L_y *= Last stable offset(LSO)*

R_y *= Remote log end offset* R_x *= Remote log start offset*

$L_z >= L_y >= Lx$ *and* $L_y >= R_y >= R_x$

Replica Manager

If RLM is configured, ReplicaManager will call RLM to assign or remove topic-partitions.

If the broker changes its state from Leader to Follower for a topic-partition and RLM is in the process of copying the segment, it will finish the copy before it relinquishes the copy for topic-partition. This might leave duplicated segments but these will be cleaned up when these segments are ready for deletion based on remote retention configs.

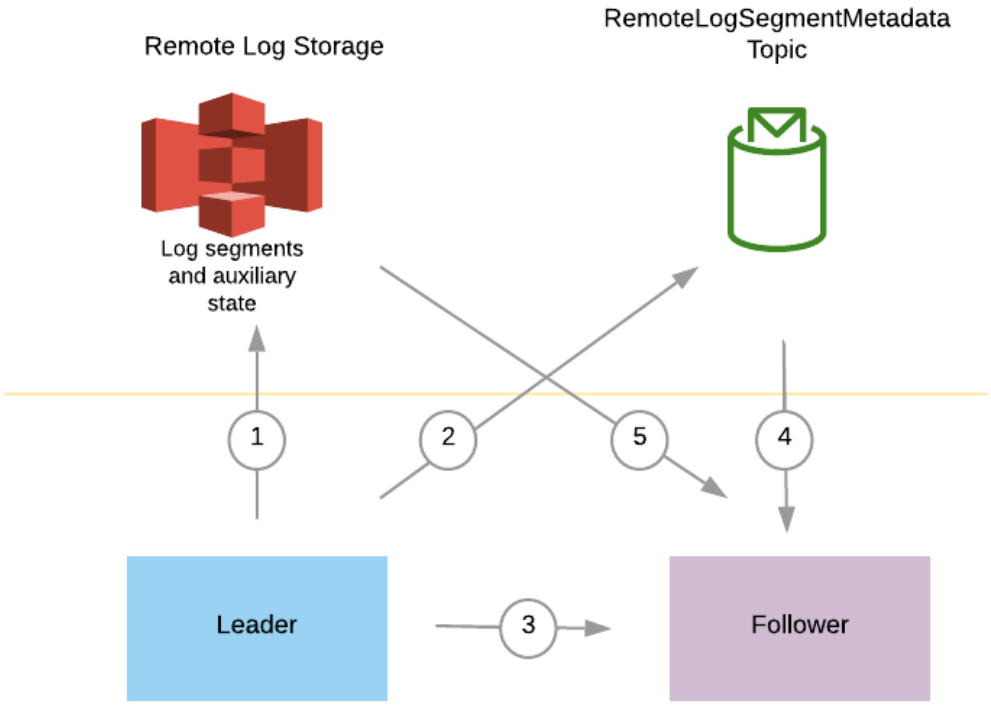
Follower Replication

Overview

Currently, followers replicate the data from the leader, and try to catch up until the log-end-offset of the leader to become in-sync replicas. Followers maintain the same log segments lineage as the leader by doing the truncation if required.

With tiered storage, followers need to maintain the same log segments lineage as the leader. Followers replicate the data that is only available on the leader's local storage. But they need to build the state like leader epoch cache and producer id snapshots for the remote segments and they also need to do truncation if required.

The below diagram gives a brief overview of the interaction between leader, follower, and remote log and metadata storage. It will be described more in detail in the next section.



1. Leader copies log segments with the auxiliary state(includes leader epoch cache and producer-id snapshots) to remote storage.
2. Leader publishes remote log segment metadata about the copied remote log segment,
3. Follower tries to fetch the messages from the leader and follows the protocol mentioned in detail in the next section.
4. Follower waits till it catches up consuming the required remote log segment metadata.
5. Follower fetches the respective remote log segment metadata to build auxiliary state.

Follower fetch protocol in detail

Leader epoch was introduced for handling possible log divergence among replicas in a few leadership change scenarios mentioned in KIP-101 and KIP-279. This is a monotonically increasing number for partition in a single leadership phase and it is stored in each message batch.

Leader epoch sequence file is maintained for each partition by each broker, and all in-sync replicas are guaranteed to have the same leader epoch history and the same log data.

Leader epoch is used to

- decide log truncation (KIP-101),
- keep consistency across replicas (KIP-279), and
- reset consumer offsets after truncation (KIP-320).

Incase of remote storage also, we should maintain log lineage and leader epochs like it is done with local storage.

Currently, followers build the auxiliary state (i.e. leader epoch sequence, producer snapshot state) when they fetch the messages from the leader by reading the message batches. Incase of tiered storage, follower finds the offset and leader epoch up to which the auxiliary state needs to be built from the leader. After which, followers start fetching the data from the leader starting from that offset. That offset can be local-log-start-offset or last-tiered-offset. Local-log-start-offset is the log start offset of the local storage. Last-tiered-offset is the offset up to which the segments are copied to remote storage. We will describe pros and cons of choosing these segments.

last-tiered-offset

- The advantage of this option is that followers can catch up quickly with the leader as the segments that are required to be fetched by followers are the segments that are not yet moved to remote storage.
- One disadvantage with this approach is that followers may have a few local segments than the leader. When that follower becomes a leader then the existing followers will truncate their logs to the leader's local log-start-offset.

local-log-start-offset

- This will honour local log retention in case of leader switches.
- It will take longer for a lagging follower to become an insync replica by catching up with the leader. One of those cases can be a new follower replica added for a partition need to start fetching from local log start offset to become an insync follower. So, this may take longer based on the local log segments available on the leader.

We prefer to go with the local log start offset as the offset from which follower starts to replicate the local log segments for the reasons mentioned above.

With tiered storage, the leader only returns the data that is still in the leader's local storage. Log segments that exist only on remote storage are not replicated to followers as those are already present in remote storage. Followers fetch offsets and truncate their local logs if needed with the current mechanism based on the leader's local-log-start-offset. This is described with several cases in detail in the next section.

When a follower fetches data for an offset which is no longer available in the leader's local storage, the leader will send a new error code `*OFFSET_MOVED_TO_TIERED_STORAGE*`. After that, follower finds the local-log-start-offset and respective leader epoch from the leader. Followers need to build the auxiliary state of the remote log segments till that offset, which are leader epochs and producer-snapshot-ids. This can be done in two ways.

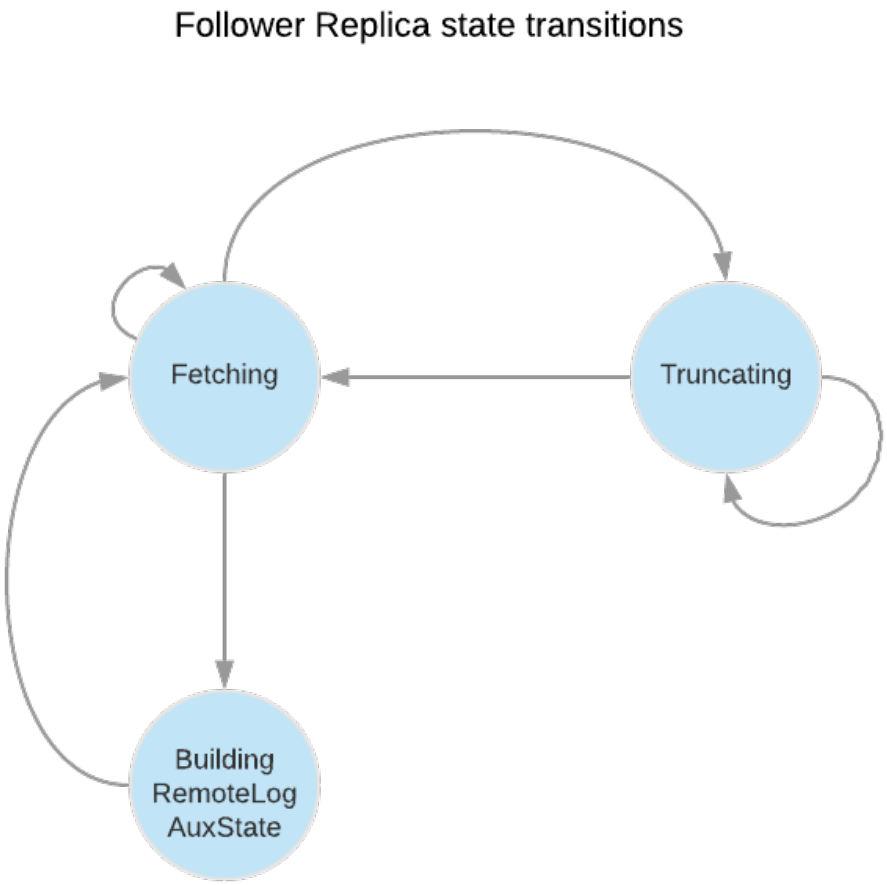
- introduce a new protocol (or API) to fetch this state from the leader partition.
- fetch this state from the remote storage.

Latter is preferred here as remote storage can have this state and it is simpler without introducing a new protocol with the leader.

This involves two steps in getting the required state of the respective log segment for the requested fetch offset.

- it should fetch the respective remote log segment metadata and
- it should fetch respective state like leader epochs from remote storage for the respective remote log segment metadata.

When shipping a log segment to remote storage, the leader broker will store the leader epoch sequence and producer id snapshot up to the end of the segment into the same remote directory (or the same remote object key prefix). These data can be used by the followers to rebuild the leader epoch sequences and producer id snapshots when needed.



So, we need to add a respective ReplicaState for building auxiliary state which can be called `*BuildingRemoteLogAuxState*`. Fetcher thread processes this state also in every run as it does for Fetching and Truncating states.

When a follower tries to fetch an offset that is no longer in the leader's local storage, the leader returns *OffsetMovedToRemoteStorage* error. Upon receiving this error, the follower will

1) Retrieve the Earliest Local Offset (ELO) and the corresponding leader epoch (ELO-LE) from the leader with a ListOffset request (timestamp = -4)

2) Truncate local log and local auxiliary state

3) Transfer from Fetching state to BuildingRemoteLogAux state

In BuildingRemoteLogAux state, the follower will

Option 1:

Repeatedly call the FetchEarliestOffsetFromLeader API from ELO-LE to the earliest leader epoch that the leader knows, and build local leader epoch cache accordingly. This option may not be very efficient when there were a lot of leadership changes. The advantage of this option is that the entire process is in Kafka, even when the remote storage is temporarily unavailable, the followers can still catch up and join ISR.

Option 2:

1) Wait for RLMM to receive remote segment information until there is a remote segment that contains the ELO-LE.

2) Fetch the leader epoch sequence and producer snapshot from remote storage (using remote storage fetcher thread pool)

3) Build the local leader epoch cache by cutting the leader epoch sequence received from remote storage to [LSO, ELO]. (LSO = log start offset)

After building the local leader epoch cache, the follower transfers back to Fetching state, and continues fetching from ELO. We preferred to go with the latter option as it can get the required state from remote storage.

Let us discuss a few cases that followers can encounter while it tries to replicate from the leader and build the auxiliary state from remote storage.

OMTS : OffsetMovedToTieredStorage

ELO : Earliest-Local-Offset

LE-x : Leader Epoch x,

HW : High Watermark

seg-a-b: a remote segment with first-offset = a and last-offset = b

LE-x, y : A leader epoch sequence entry indicates leader-epoch x starts from offset y

Follower fetch scenarios(including truncation cases)

Scenario 1: new empty follower

A broker is added to the cluster and assigned as a replica for a partition. This broker will not have any local data as it has just become a follower for the first time. It will try to fetch the offset 0 from the leader. If that offset does not exist on the leader, the follower will receive the OFFSET_MOVED_TO_TIERED_STORAGE error. The follower will then send a ListOffset request with timestamp = EARLIEST_LOCAL_TIMESTAMP, and will receive the offset of the leader's earliest local message.

The follower will need to build the state until that offset before it starts to fetch from the leader's local storage.

step 1:

Fetch remote segment info, and rebuild leader epoch sequence.

Broker A (Leader)	Broker B (Follower)	Remote Storage	RL metadata storage

3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 6: msg 6 LE-2 7: msg 7 LE-3 (HW) leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	1. Fetch LE-1, 0 2. Receives OMTS 3. Receives ELO 3, LE-1 4. Fetch remote segment info and build local leader epoch sequence until ELO leader_epochs LE-0, 0 LE-1, 3	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-5, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 5	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-3-5, uuid-2 segment epochs LE-1, 3 LE-2, 5

step 2:

continue fetching from the leader

Broker A (Leader)	Broker B (Follower)	Remote Storage	RL metadata storage
3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 6: msg 6 LE-2 7: msg 7 LE-3 (HW) leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	Fetch from ELO to HW 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 6: msg 6 LE-2 7: msg 7 LE-3 (HW) leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-5, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 5	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-3-5, uuid-2 segment epochs LE-1, 3 LE-2, 5

Scenario 2: out-of-sync follower catching up

A follower is trying to catch up, and the segment has moved to tiered storage. It involves two cases like whether the local segment exists or not.

2.1 Local segment exists and the latest local offset is larger than the earliest-local-offset of the leader

In this case, followers fetch like earlier as the local segments exist. There will not be any changes for this case.

2.2 Local segment does not exist, or the latest local offset is smaller than ELO of the leader

In this case, local segments might have already been deleted because of the local retention settings, or the follower has been offline for a very long time. The follower receives *OFFSET_MOVED_TO_TIERED_STORAGE* error while trying to fetch the desired offset. The follower has to truncate all the local log segments because we know the data already expired on the leader.

step 1:

An out-of-sync follower (broker B) has local data up to offset 3

Broker A (Leader)	**Broker B (Follower)**	**Remote Storage**	**RL metadata storage**
0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 6: msg 6 LE-2 7: msg 7 LE-3 8: msg 8 LE-3 9: msg 9 LE-3 (HW) leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 leader_epochs LE-0, 0 LE-1, 3 1. Because the latest leader epoch in the local storage (LE-1) does not equal the current leader epoch (LE-3). The follower starts from the Truncating state. 2. fetchLeaderEpochEndOffsets(LE-1) returns 5, which is larger than the latest local offset. With the existing truncation logic, the local log is not truncated and it moves to Fetching state.	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-5, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 5	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-3-5, uuid-2 segment epochs LE-1, 3 LE-2, 5

step 2:

Local segments on the leader are deleted because of retention, and then the follower starts trying to catch up with the leader.

Broker A (Leader)	**Broker B (Follower)**	**Remote Storage**	**RL metadata storage**

9: msg 9 LE-3 10: msg 10 LE-3 11: msg 11 LE-3 (HW) [segments till offset 8 were deleted] leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 leader_epochs LE-0, 0 LE-1, 3 <Fetch State> 1. Fetch from leader LE-1, 4 2. Receives OMTS, truncate local segments. 3. Fetch ELO, Receives ELO 9, LE-3 and moves to *BuildingRemoteLogAux state*	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-5, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 5 Seg 6-8, uuid-3, LE-3 log: 6: msg 6 LE-2 7: msg 7 LE-3 8: msg 8 LE-3 epochs: LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-3-5, uuid-2 segment epochs LE-1, 3 LE-2, 5 seg-6-8, uuid-3 segment epochs LE-2, 5 LE-3, 7

step 3:

After deleting the local data, this case becomes the same as scenario 1.

Broker A (Leader)	**Broker B (Follower)**	**Remote Storage**	**RL metadata storage**

9: msg 9 LE-3 10: msg 10 LE-3 11: msg 11 LE-3 (HW) [segments till offset 8 were deleted] leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	1. follower rebuilds leader epoch sequence up to LE-3 using remote segment metadata and remote data leader_epochs LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7 2. follower continue fetching from the leader from ELO (9, LE-3) 9: msg 9 LE-3 10: msg 10 LE-3 11: msg 11 LE-3 (HW)	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-5, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 5 Seg 6-8, uuid-3, LE-3 log: 6: msg 6 LE-2 7: msg 7 LE-3 8: msg 8 LE-3 epochs: LE-0, 0 LE-1, 3 LE-2, 5 LE-3, 7	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-3-5, uuid-2 segment epochs LE-1, 3 LE-2, 5 seg-6-8, uuid-3 segment epochs LE-2, 5 LE-3, 7

Scenario 3: Multiple hard failures (Scenario 2 of KIP-101)

Step 1:

Broker A (Leader)	Broker B	Remote Storage	RL metadata storage
0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 (HW) leader_epochs LE-0, 0	0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 (HW) leader_epochs LE-0, 0	seg-0-1: log: 0: msg 0 LE-0 1: msg 1 LE-0 epoch: LE-0, 0	seg-0-1, uuid-1 segment epochs LE-0, 0

Broker A has shipped its 1st log segment to remote storage.

Step 2:

Both broker A and broker B crashed at the same time. Some messages (msg 1 and msg 2) on broker B were not synced to the hard disk, and were lost.

In this case, it is acceptable to lose data, but we have to keep the same behaviour as described in the KIP-101.

Broker A (stopped)	Broker B (Leader)	Remote Storage	RL metadata storage

0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 (HW) leader_epochs LE-0, 0	0: msg 0 LE-0 (HW) 1: msg 3 LE-1 leader_epochs LE-0, 0 LE-1, 1	seg-0-1: log: 0: msg 0 LE-0 1: msg 1 LE-0 epoch: LE-0, 0	seg-0-1, uuid-1 segment epochs LE-0, 0

After restart, B losses message 1 and 2. B becomes the new leader, and receives a new message 3 (LE-1, offset 1).

(Note: This may not be technically an unclean-leader-election, because B may have not been removed from ISR because both of the 2 brokers crashed at the same time.)

Step 3:

After restart, broker A truncates offset 1 and 2 (LE-0), and receives the new message (LE-1, offset 1).

Broker A (follower)	Broker B (Leader)	Remote Storage	RL metadata storage
0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 1: msg 3 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	0: msg 0 LE-0 1: msg 3 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	seg-0-1: log: 0: msg 0 LE-0 1: msg 1 LE-0 epoch: LE-0, 0	seg-0-1, uuid-1 segment epochs LE-0, 0

Step 4:

Broker A (follower)	Broker B (Leader)	Remote Storage	RL metadata storage
0: msg 0 LE-0 1: msg 3 LE-1 2: msg 4 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	0: msg 0 LE-0 1: msg 3 LE-1 2: msg 4 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	seg-0-1: log: 0: msg 0 LE-0 1: msg 1 LE-0 epoch: LE-0, 0 seg-1-1 log: 1: msg 3 LE-1 epoch: LE-0, 0 LE-1, 1	seg-0-1, uuid-1 segment epochs LE-0, 0 seg-1-1, uuid-2 segment epochs LE-1, 1

A new message (message 4) is received. The 2nd segment on broker B (seg-1-1) is shipped to remote storage.

Consider the local segments up to offset 2 are deleted on both brokers:

A consumer fetches offset 0, LE-0. According to the local leader epoch cache, offset 0 LE-0 is valid. So, the broker returns message 0 from remote segment 0-1.

A pre-KIP-320 consumer fetches offset 1, without leader epoch info. According to the local leader epoch cache, offset 1 belongs to LE-1. So, the broker returns message 3 from remote segment 1-1, rather than the LE-0 offset 1 message 1 in seg-0-1.

A consumer fetches offset 2 LE-0 is fenced (KIP-320).

A consumer fetches offset 1 LE-1 receives message 3 from remote segment 1-1.

Scenario 4: unclean leader election including truncation.

Step 1:

Broker A (Leader)	Broker B (out-of-sync)	Remote Storage	RL metadata storage
0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-0 (HW) leader_epochs LE-0, 0	0: msg 0 LE-0 (HW) leader_epochs LE-0, 0	seg 0-2: log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epoch: LE-0, 0	seg-0-2, uuid-1 segment epochs LE-0, 0

Step 2:

Broker A (Stopped)	Broker B (Leader)	Remote Storage	RL metadata storage
	0: msg 0 LE-0 1: msg 4 LE-1 2: msg 5 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	seg 0-2: log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epoch: LE-0, 0 seg 0-1: 0: msg 0 LE-0 1: msg 4 LE-1 epoch: LE-0, 0 LE-1, 1	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-0-1, uuid-2 segment epochs LE-0, 0 LE-1, 1

Broker A stopped, an out-of-sync replica (broker B) became the new leader. With unclean-leader-election, it's acceptable to lose data, but we have to make sure the existing Kafka behaviour is not changed.

We assume min.in_sync = 1 in this example.

Broker B ships its local segment (seg-0-1) to remote storage, after the highwater mark is moved to 2 (message 5).

Step 3:

Broker A (Stopped)	Broker B (Leader)	Remote Storage	RL metadata storage
	2: msg 5 LE-1 (HW) leader_epochs LE-0, 0 LE-1, 1	seg 0-2: log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epoch: LE-0, 0 seg 0-1: 0: msg 0 LE-0 1: msg 4 LE-1 epoch: LE-0, 0 LE-1, 1	seg-0-2, uuid-1 segment epochs LE-0, 0 seg-0-1, uuid-2 segment epochs LE-0, 0 LE-1, 1

The 1st local segment on broker B expired.

A consumer fetches offset 0 LE-0 receives message 0 (LE-0, offset 0). This message can be served from either remote segment seg-0-2 or seg-0-1.

A pre-KIP-320 consumer fetches offset 1. The broker finds offset 1 belongs to leader epoch 1. So, it returns message 4 (LE-1, offset 1) to the consumer, rather than message 1 (LE-0, offset 1).

A post-KIP-320 consumer fetches offset 1 LE-1 receives message 4 (LE-1, offset 1) from remote segment 0-1.

A consumer fetches offset 2 LE-0 is fenced (KIP-320).

Scenario 5: log divergence in remote storage - unclean leader election

step 1

Broker A (Leader)	**Broker B**	**Remote Storage**	**Remote Segment Metadata**
0: msg 0 LE-0	0: msg 0 LE-0	seg-0-3	seg-0-3, uuid1
1: msg 1 LE-0	1: msg 1 LE-0	log:	segment epochs
2: msg 2 LE-0	leader_epochs	0: msg 0 LE-0	LE-0, 0
3: msg 3 LE-0	LE-0, 0	1: msg 1 LE-0	
4: msg 4 LE-0 (HW)		2: msg 2 LE-0	
leader_epochs	broker B is out-of-sync	3: msg 3 LE-0	
LE-0, 0		epoch:	
broker A shipped one segment to remote storage		LE0, 0	

step 2

An out-of-sync broker B becomes the new leader, after broker A is down. (unclean leader election)

Broker A (stopped)	**Broker B (Leader)**	Remote Storage	**RL metadata storage**
0: msg 0 LE-0	0: msg 0 LE-0	seg-0-3	seg-0-3, uuid1
1: msg 1 LE-0	1: msg 1 LE-0	log:	segment epochs
2: msg 2 LE-0	2: msg 4 LE-1	0: msg 0 LE-0	LE-0, 0
3: msg 3 LE-0	3: msg 5 LE-1	1: msg 1 LE-0	
4: msg 4 LE-0	4: msg 6 LE-1	2: msg 2 LE-0	seg-0-3, uuid2
leader_epochs	leader_epochs	3: msg 3 LE-0	segment epochs
LE-0, 0	LE-0, 0	epoch:	LE-0, 0
	LE-1, 2	LE-0, 0	LE-1, 2
		Seg-0-3	
	After becoming the new leader, B received several new messages, and shipped one segment to remote storage.	0: msg 0 LE-0	
		1: msg 1 LE-0	
		2: msg 4 LE-1	
		3: msg 5 LE-1	
		epoch:	
		LE-0, 0	
		LE-1, 2	

step 3

Broker B is down. Broker A restarted without knowing LE-1. (another unclean leader election)

Broker A (Leader)	Broker B (stopped)	Remote Storage	RL metadata storage
0: msg 0 LE-0	0: msg 0 LE-0	seg-0-3	seg-0-3, uuid1
1: msg 1 LE-0	1: msg 1 LE-0	log:	segment epochs
2: msg 2 LE-0	2: msg 4 LE-1	0: msg 0 LE-0	LE-0, 0
3: msg 3 LE-0	3: msg 5 LE-1	1: msg 1 LE-0	
4: msg 4 LE-0	4: msg 6 LE-1	2: msg 2 LE-0	seg-0-3, uuid2
5: msg 7 LE-2	leader_epochs	3: msg 3 LE-0	segment epochs
6: msg 8 LE-2	LE-0, 0	epoch:	LE-0, 0
leader_epochs	LE-1, 2	LE-0, 0	LE-1, 2
LE-0, 0		seg-0-3	
LE-2, 5		0: msg 0 LE-0	seg-4-5, uuid3
1. Broker A receives two new messages in LE-2		1: msg 1 LE-0	segment epochs
2. Broker A ships seg-4-5 to remote storage		2: msg 4 LE-1	LE-0, 0
		3: msg 5 LE-1	LE-2, 5
		epoch:	
		LE-0, 0	
		LE-1, 2	
		seg-4-5	
		epoch:	
		LE-0, 0	
		LE-2, 5	

step 4

Broker B reimaged and lost all the local data

Broker A (Leader)	Broker B (started, follower)	Remote Storage	RL metadata storage

6: msg 8 LE-2 leader_epochs LE-0, 0 LE-2, 5	1. Broker B fetches offset 0, and receives OMTS error. 2. Broker B receives ELO=6, LE-2 3. in BuildingRemoteLogAux state, broker B finds seg-4-5 has LE-2. So, it builds local LE cache from seg-4-5: leader_epochs LE-0, 0 LE-2, 5 4. Broker B continue fetching from local messages from ELO 6, LE-2 5. Broker B joins ISR	seg-0-3 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-0 epoch: LE-0, 0 seg-0-3 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 4 LE-1 3: msg 5 LE-1 epoch: LE-0, 0 LE-1, 2 seg-4-5 epoch: LE-0, 0 LE-2, 5	seg-0-3, uuid1 segment epochs LE-0, 0 seg-0-3, uuid2 segment epochs LE-0, 0 LE-1, 2 seg-4-5, uuid3 segment epochs LE-0, 0 LE-2, 5

A consumer fetches offset 3, LE-1 from broker B will be fenced.

A pre-KIP-320 consumer fetches offset 2 from broker B will get msg 2 (offset 2, LE-0).

Follower to leader transition

A follower can be considered as a leader by the controller based on its replica configuration. When a follower becomes a leader it needs to find out the offset from which the segments to be copied to remote storage. This is found by traversing from the latest leader epoch from leader epoch history and find the highest offset of a segment with that epoch copied into remote storage. If it can not find an entry then it checks for the previous leader epoch till it finds an entry, If there are no entries till the earliest leader epoch in leader epoch cache then it starts copying the segments from the earliest epoch entry's offset.

Step 1:

Broker A (Leader)	Broker B (Follower)	Remote Storage	RL metadata storage

0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-1 6: msg 6 LE-2 (HW) 7: msg 7 LE-2 8: msg 8 LE-2 leader_epochs LE-0, 0 LE-1, 3 LE-2, 6	0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-1 6: msg 6 LE-2 (HW) leader_epochs LE-0, 0 LE-1, 3 LE-2, 6	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg 3-4, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 epochs: LE-0, 0 LE-1, 3	seg-0-2, uuid-1 Segment epochs LE-0, 0 seg-3-4, uuid-2 Segment epochs LE-1, 3

Step 2:

Broker A is crashed/stopped and Broker B became a leader. It checks from leader epoch-2 whether there are any segments and it traverses back till it finds a segment for the leader epoch. In this case, it finds offset 4 for leader epoch-1 from RLMM. It needs to copy segments containing offset 5. So, it starts copying from the “seg-4-6” segment.

Broker A (Stopped)	**Broker B (Leader)**	**Remote Storage**	**RL metadata storage**

0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-1 6: msg 6 LE-2 (HW) 7: msg 7 LE-2 8: msg 8 LE-2 leader_epochs LE-0, 0 LE-1, 3 LE-2, 6	0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 3: msg 3 LE-1 4: msg 4 LE-1 5: msg 5 LE-1 6: msg 6 LE-2 (HW) 7: msg 8 LE-3 leader_epochs LE-0, 0 LE-1, 3 LE-2, 6 LE-3, 7	seg-0-2, uuid-1 log: 0: msg 0 LE-0 1: msg 1 LE-0 2: msg 2 LE-0 epochs: LE-0, 0 seg-3-4, uuid-2 log: 3: msg 3 LE-1 4: msg 4 LE-1 epochs: LE-0, 0 LE-1, 3 Seg-4-6, uuid-3 4: msg 4 LE-1 5: msg 5 LE-1 6: msg 6 LE-2 epochs: LE-0, 0 LE-1, 3 LE-2, 6	seg-0-2, uuid-1 Segment epochs LE-0, 0 seg-3-4, uuid-2 Segment epochs LE-1, 3 seg-4-6, uuid-3 Segment epochs LE-1, 3 LE-2, 6

Transactional support

RemoteLogManager copies transaction index and producer-id-snapshot along with the respective log segment earlier to last-stable-offset. This is used by the followers to return aborted transactions in fetch requests with isolation level as READ_COMMITTED.

Consumer Fetch Requests

For any fetch requests, ReplicaManager will proceed with making a call to readFromLocalLog, if this method returns OffsetOutOfRange exception it will delegate the read call to RemoteLogManager. More details are explained in the RLM/RSM tasks section. If the remote storage is not available then it will throw a new error TIERED_STORAGE_NOT_AVAILABLE.

Other APIs

DeleteRecords

There is no change in the semantics of this API. It deletes records until the given offset if possible. This is equivalent to updating logStartOffset of the partition log with the given offset if it is greater than the current log-start-offset and it is less than or equal to high-watermark. If needed, it will clean remote logs asynchronously after updating the log-start-offset of the log. RLMTask for leader partition periodically checks whether there are remote log segments earlier to logStartOffset and the respective remote log segment metadata and data are deleted by using RLMM and RSM.

ListOffsets

ListOffsets API gives the offset(s) for the given timestamp either by looking into the local log or remote log time indexes.

If the target timestamp is

ListOffsetRequest.EARLIEST_TIMESTAMP (value as -2) returns logStartOffset of the log.

ListOffsetRequest.LATEST_TIMESTAMP(value as-1) returns log-stable-offset or log-end-offset based on the isolation level in the request.

This API is enhanced with supporting new target timestamp value as -4 which is called EARLIEST_LOCAL_TIMESTAMP. There will not be any new fields added in request and response schemes but there will be a version bump to indicate the version update. This request is about the offset that the followers should start fetching to replicate the local logs. It represents the log-start-offset available in the local log storage which is also called as local-log-start-offset. All the records earlier to this offset can be considered as copied to the remote storage. This is used by follower replicas to avoid fetching records that are already copied to remote tier storage.

When a follower replica needs to fetch the earliest messages that are to be replicated then it sends a request with the target timestamp as EARLIEST_LOCAL_TIMESTAMP.

For timestamps >= 0, it returns the first message offset whose timestamp is >= to the given timestamp in the request. That means it checks in remote log time indexes first, after which local log time indexes are checked.

LeaderAndIsr

This is received by RLM to register for new leaders so that the data can be copied to the remote storage. RLMM will also register the respective metadata partitions for the leader/follower partitions if they are not yet subscribed.

Stopreplica

RLM receives a callback and unassigns the partition for leader/follower task, If the delete option is enabled then the leader will stop RLM task and stop processing. The controller will not allow topic with the same name to be created till all the segments are cleaned up from remote storage.

It was discussed in the community earlier for adding UUID to represent a topic along with the name as part of KIP-516. This enhancement will be useful to make the deletion of topic partitions in remote storage asynchronously without blocking the creation of topic with the same name even though all the segments are not deleted in remote storage.

OffsetForLeaderEpoch

Look into leader epoch checkpoint cache. This is stored in tiered storage and it may be fetched by followers from tiered storage as part of the fetch protocol.

LogStartOffset

LogStartOffset of a topic can point to either of local segment or remote segment but it is initialised and maintained in the Log class like now. This is already maintained in `Log` class while loading the logs and it can also be fetched from RemoteLogMetadataManager.

There are no changes with other protocol APIs because of tiered storage.

RLM/RSM tasks and thread pools

Remote storage (e.g. HDFS/S3/GCP) is likely to have higher I/O latency and lower availability than local storage.

When the remote storage becoming temporarily unavailable (up to several hours) or having high latency (up to minutes), Kafka should still be able to operate normally. All the Kafka operations (produce, consume local data, create/expand topics, etc.) that do not rely on remote storage should not be impacted. The consumers that try to consume the remote data should get reasonable errors, when remote storage is unavailable or the remote storage requests timeout.

To achieve this, we have to handle remote storage operations in dedicated threads pools, instead of Kafka I/O threads and fetcher threads.

1. Remote Log Manager (RLM) Thread Pool

RLM maintains a list of the topic-partitions it manages. The list is updated in Kafka I/O threads, when topic-partitions are added to / removed from RLM. Each topic-partition in the list is assigned a scheduled processing time. The RLM thread pool processes the topic-partitions that the "scheduled processing time" is less than or equal to the current time.

When a new topic-partition is assigned to the broker, the topic-partition is added to the list, with scheduled processing time = 0, which means the topic-partition has to be processed immediately, to retrieve information from remote storage.

After a topic-partition is successfully processed by the thread pool, it's scheduled processing time is set to (now() + remote.log.manager.task.interval.ms). remote.log.manager.task.interval.ms can be configured in broker config file.

If the process of a topic-partition is failed due to remote storage error, it follows retry backing off algorithm with intiial retry interval as `remote.log.manager.task.retry.interval.ms`, max backoff as `remote.log.manager.task.retry.backoff.max.ms`, and jitter as `remote.log.manager.task.retry.jitter`. You can see more details about the exponential backoff algorithm here.

When a topic-partition is unassigned from the broker, the topic-partition is not currently processed by the thread pool, the topic-partition is directly removed from the list; otherwise, the topic-partition is marked as "deleted", and will be removed after the current process is done.

Each thread in the thread pool processes one topic-partition at a time in the following steps:

Copy log segments to remote storage (leader)

Copy the log segment files that are

- inactive and

- the offset range is not covered completely by the segments on the remote storage and

- those segments have the last offset < last-stable-offset of the partition.

If multiple log segment files are ready, they are copied to remote storage one by one, from the earliest to the latest. It generates a universally unique *RemoteLogSegmentId for each segment, it calls RLMM.putRemoteLogSegmentData(RemoteLogSegmentMetadata remoteLogSegmentMetadata)* and it invokes *copyLogSegment(RemoteLogSegmentMetadata remoteLogSegmentMetadata, LogSegmentData logSegmentData)* on RSM. If it is successful then it calls RLMM.putRemoteLogSegmentData with the updated state in RemoteLogSegmentMetadata.

Handle expired remote segments (leader)

RLM leader computes the log segments to be deleted based on the remote retention config. It updates the earliest offset for the given topic partition in RLMM. It gets all the remote log segment ids and removes them from remote storage using RemoteStorageManager. It also removes respective metadata using RemoteLogMetadataManager.

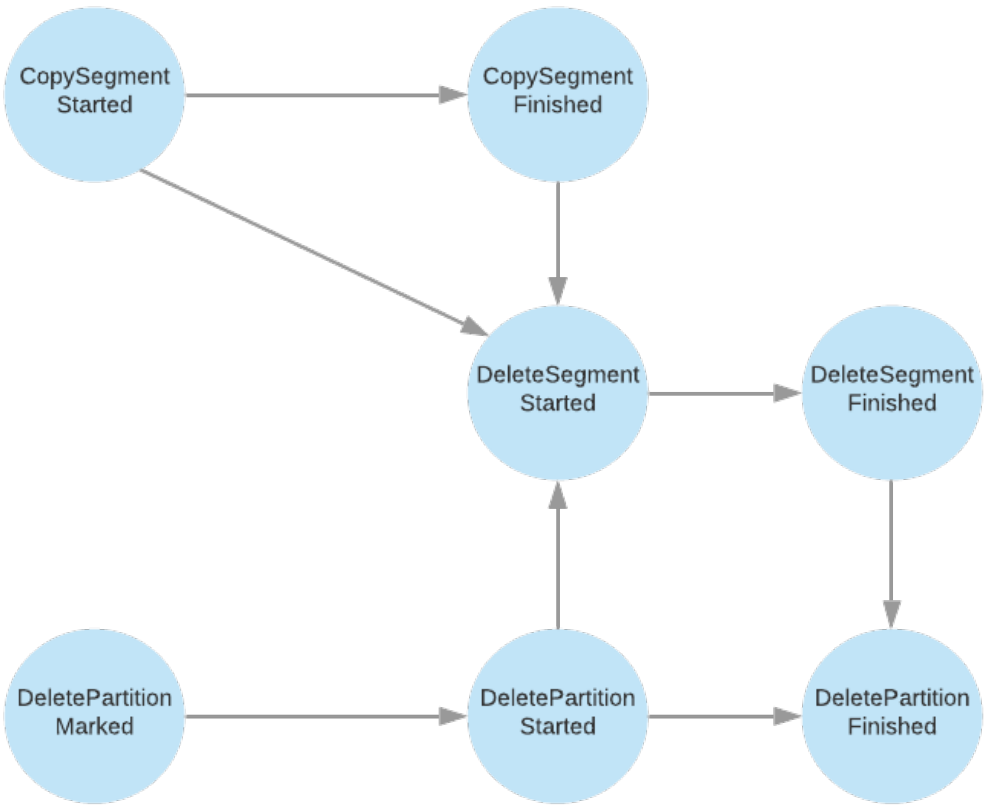
2. Remote Storage Fetcher Thread Pool

When handling consumer fetch request, if the required offset is in remote storage, the request is added into "*RemoteFetchPurgatory*", to handle timeout. *RemoteFetchPurgatory* is an instance of *kafka.server.DelayedOperationPurgatory*, and is similar to the existing produce/fetch purgatories. At the same time, the request is put into the task queue of "remote storage fetcher thread pool".

Each thread in the thread pool processes one remote fetch request at a time. The remote storage fetch thread will

1. find out the corresponding RemoteLogSegmentId from RLMM and startPosition and endPosition from the offset index.
2. try to build Records instance data fetched from *RSM.fetchLogSegmentData(RemoteLogSegmentMetadata remoteLogSegmentMetadata, Long startPosition, Long endPosition)*
    a. if success, RemoteFetchPurgatory will be notified to return the data to the client
    b. if the remote segment file is already deleted, RemoteFetchPurgatory will be notified to return an error to the client.
    c. if the remote storage operation failed (remote storage is temporarily unavailable), the operation will be retried with Exponential Back-Off, until the original consumer fetch request timeout.

Remote Log Metadata State transitions



COPY_SEGMENT_STARTED - This state indicates that the segment copying to remote storage is started but not yet finished.

COPY_SEGMENT_FINISHED - This state indicates that the segment copying to remote storage is finished.

The leader broker copies the log segments to the remote storage and puts the remote log segment metadata with the state as "*COPY_SEGMENT_STARTED*" and updates the state as "*COPY_SEGMENT_FINISHED*" once the copy is successful.

DELETE_SEGMENT_STARTED - This state indicates that the segment deletion is started but not yet finished.

DELETE_SEGMENT_FINISHED - This state indicates that the segment is deleted successfully.

Leader partitions publish both the above delete segment events when remote log retention is reached for the respective segments. Remote Partition Removers also publish these events when a segment is deleted.

DELETE_PARTITION_MARKED - This is published when a topic/partition is deleted by the controller. This partition is marked for delete by the controller. That means, all its remote log segments are eligible for deletion so that remote partition removers can start deleting them.

DELETE_PARTITION_STARTED - This state indicates that the partition deletion is started but not yet finished.

DELETE_PARTITION_FINISHED - This state indicates that the partition is deleted successfully.

Remote Partition Removers also publish these events when a partition is deleted.

When a partition is deleted, the controller updates its state in RLMM with DELETE_PARTITION_MARKED and it expects RLMM will have a mechanism to clean up the remote log segments. This process for default RLMM is described in detail here.

RemoteLogMetadataManager implemented with an internal topic

Metadata of remote log segments are stored in an internal non compact topic called `__*remote_log_metadata*`. This topic can be created with default partitions count as 50. Users can configure the partitions count and replication factor etc as mentioned in the config section.

In this design, *RemoteLogMetadataManager(RLMM)* is responsible for storing and fetching remote log segment metadata. It provides

- Storing remote log segment metadata for a partition's log segment
- Fetching remote log segment metadata for an offset and leader epoch.
- Register a topic partition to build cache for remote log segment metadata by reading from remote log segment metadata topic

RemoteLogMetadataManager(RLMM) mainly has the below components

- Cache
- Producer
- Consumer

Remote log metadata topic partition for a given user topic-partition is:

Utils.toPositive(Utils.murmur2(tp.toString().getBytes(StandardCharsets.UTF_8))) % no_of_remote_log_metadata_topic_partitions

RLMM registers the topic partitions that the broker is either a leader or a follower. These topic partitions include the remote log metadata topic partitions also.

RLMM maintains metadata cache by subscribing to the respective remote log metadata topic partitions. Whenever a topic partition is reassigned to a new broker and *RLMM* on that broker is not subscribed to the respective remote log metadata topic partition then it will subscribe to the respective remote log metadata topic partition and adds all the entries to the cache. So, in the worst case, *RLMM* on a broker may be consuming from most of the remote log metadata topic partitions. In the initial version, we will have a file-based cache for all the messages that are already consumed by this instance and it will load in-memory whenever RLMM is started. This cache is maintained in a separate file for each of the topic partitions. This will allow us to commit offsets of the partitions that are already read. Committed offsets can be stored in a local file to avoid reading the messages again when a broker is restarted.

RLMM segment overhead:

Topic partition's topic-id : uuid : 2 longs.

remoteLogSegmentId : uuid : 2 longs.

remoteLogSegmentMetadata : 5 longs + 1 int +1 byte + ~3 epochs(approx avg)

It has leader epochs in-memory which will be much less.

On avg: 10 longs : 10 * 8 = 80 *(other overhead 1.25) = 100 bytes

When a segment is rolled on a broker per sec.

retention as 30days : 60*60*24*30 ~ 2.6MM

2.6MM segments would take ~ 260MB. (This is 1% in our production env)

This overhead is not that significant as brokers may be using several GBs of memory.

We can also have a lazy load approach by keeping only minimal in-memory entries like offset, epoch, uuid, and entry position in the file. When it is needed we can access it by using the entry position in the file.

Message Format

RLMM instance on broker publishes the message to the topic with key as null and value with the below format.

type : Represents the value type. This value is 'apikey' as mentioned in the schema. Its type is 'byte'.
version : the 'version' number of the type as mentioned in the schema. Its type is 'byte'.
data : record payload in kafka protocol message format, the schema is given below.

Both type and version are added before the data is serialized into record value. Schema can be evolved by adding a new version with the respective changes. A new type can also be supported by adding the respective type and its version.

Schema

```
{
  "apiKey": 0,
  "type": "data",
  "name": "RemoteLogSegmentMetadataRecord",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    {
      "name": "RemoteLogSegmentId",
      "type": "RemoteLogSegmentIdEntry",
      "versions": "0+",
      "about": "Unique representation of the remote log segment",
      "fields": [
        {
          "name": "TopicIdPartition",
          "type": "TopicIdPartitionEntry",
          "versions": "0+",
          "about": "Represents unique topic partition",
          "fields": [
            {
              "name": "Name",
              "type": "string",
              "versions": "0+",
              "about": "Topic name"
            },
            {
              "name": "Id",
              "type": "uuid",
              "versions": "0+",
              "about": "Unique identifier of the topic"
            },
            {
              "name": "Partition",
              "type": "int32",
              "versions": "0+",
              "about": "Partition number"
            }
          ]
        },
        {
          "name": "Id",
          "type": "uuid",
          "versions": "0+",
          "about": "Unique identifier of the remote log segment"
        }
      ]
    },
    {
      "name": "StartOffset",
      "type": "int64",
      "versions": "0+",
      "about": "Start offset  of the segment."
    },
```

```
    {
      "name": "EndOffset",
      "type": "int64",
      "versions": "0+",
      "about": "End offset  of the segment."
    },
    {
      "name": "LeaderEpoch",
      "type": "int32",
      "versions": "0+",
      "about": "Leader epoch from which this segment instance is created or updated"
    },
    {
      "name": "MaxTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Maximum timestamp with in this segment."
    },
    {
      "name": "EventTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Event timestamp of this segment."
    },
    {
      "name": "SegmentLeaderEpochs",
      "type": "[]SegmentLeaderEpochEntry",
      "versions": "0+",
      "about": "Leader epoch cache.",
      "fields": [
        {
          "name": "LeaderEpoch",
          "type": "int32",
          "versions": "0+",
          "about": "Leader epoch"
        },
        {
          "name": "Offset",
          "type": "int64",
          "versions": "0+",
          "about": "Start offset for the leader epoch"
        }
      ]
    },
    {
      "name": "SegmentSizeInBytes",
      "type": "int32",
      "versions": "0+",
      "about": "Segment size in bytes"
    },
    {
      "name": "RemoteLogSegmentState",
      "type": "int8",
      "versions": "0+",
      "about": "State of the remote log segment"
    }
  ]
}


{
  "apiKey": 1,
  "type": "data",
  "name": "RemoteLogSegmentMetadataRecordUpdate",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    {
      "name": "RemoteLogSegmentId",
      "type": "RemoteLogSegmentIdEntry",
      "versions": "0+",
```

```
      "about": "Unique representation of the remote log segment",
      "fields": [
        {
          "name": "TopicIdPartition",
          "type": "TopicIdPartitionEntry",
          "versions": "0+",
          "about": "Represents unique topic partition",
          "fields": [
            {
              "name": "Name",
              "type": "string",
              "versions": "0+",
              "about": "Topic name"
            },
            {
              "name": "Id",
              "type": "uuid",
              "versions": "0+",
              "about": "Unique identifier of the topic"
            },
            {
              "name": "Partition",
              "type": "int32",
              "versions": "0+",
              "about": "Partition number"
            }
          ]
        },
        {
          "name": "Id",
          "type": "uuid",
          "versions": "0+",
          "about": "Unique identifier of the remote log segment"
        }
      ]
    },
    {
      "name": "LeaderEpoch",
      "type": "int32",
      "versions": "0+",
      "about": "Leader epoch from which this segment instance is created or updated"
    },
    {
      "name": "EventTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Event timestamp of this segment."
    },
    {
      "name": "RemoteLogSegmentState",
      "type": "int8",
      "versions": "0+",
      "about": "State of the remote segment"
    }
  ]
}



{
  "apiKey": 2,
  "type": "data",
  "name": "RemotePartitionDeleteMetadataRecord",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    {
      "name": "TopicIdPartition",
      "type": "TopicIdPartitionEntry",
      "versions": "0+",
      "about": "Represents unique topic partition",
```

```
      "fields": [
        {
          "name": "Name",
          "type": "string",
          "versions": "0+",
          "about": "Topic name"
        },
        {
          "name": "Id",
          "type": "uuid",
          "versions": "0+",
          "about": "Unique identifier of the topic"
        },
        {
          "name": "Partition",
          "type": "int32",
          "versions": "0+",
          "about": "Partition number"
        }
      ]
    },
    {
      "name": "Epoch",
      "type": "int32",
      "versions": "0+",
      "about": "Epoch (controller or leader) from which this event is created. DELETE_PARTITION_MARKED is sent
by the controller. DELETE_PARTITION_STARTED and DELETE_PARTITION_FINISHED are sent by remote log metadata topic
partition leader."
    },
    {
      "name": "EventTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Event timestamp of this segment."
    },
    {
      "name": "RemotePartitionDeleteState",
      "type": "int8",
      "versions": "0+",
      "about": "Deletion state of the remote partition"
    }
  ]
}

package org.apache.kafka.server.log.remote.storage;
...
/**
 * It indicates the deletion state of the remote topic partition. This will be based on the action executed on
this
 * partition by the remote log service implementation.
 */
public enum RemotePartitionDeleteState {

    /**
     * This is used when a topic/partition is determined to be deleted by controller.
     * This partition is marked for delete by controller. That means, all its remote log segments are eligible
for
     * deletion so that remote partition removers can start deleting them.
     */
    DELETE_PARTITION_MARKED((byte) 0),

    /**
     * This state indicates that the partition deletion is started but not yet finished.
     */
    DELETE_PARTITION_STARTED((byte) 1),

    /**
     * This state indicates that the partition is deleted successfully.
     */
    DELETE_PARTITION_FINISHED((byte) 2);
...
```

```
}


package org.apache.kafka.server.log.remote.storage;
...
/**
 * It indicates the state of the remote log segment or partition. This will be based on the action executed on
this
 * segment or partition by the remote log service implementation.
 * <p>
 */
public enum RemoteLogSegmentState {

    /**
     * This state indicates that the segment copying to remote storage is started but not yet finished.
     */
    COPY_SEGMENT_STARTED((byte) 0),

    /**
     * This state indicates that the segment copying to remote storage is finished.
     */
    COPY_SEGMENT_FINISHED((byte) 1),

    /**
     * This state indicates that the segment deletion is started but not yet finished.
     */
    DELETE_SEGMENT_STARTED((byte) 2),

    /**
     * This state indicates that the segment is deleted successfully.
     */
    DELETE_SEGMENT_FINISHED((byte) 3),
...
}
```

Configs

remote.log.metadata.topic.replication.factor	Replication factor of the topic Default: 3
remote.log.metadata.topic.num.partitions	No of partitions of the topic Default: 50
remote.log.metadata.topic.retention.ms	Retention of the topic in milli seconds. Default: -1, that means unlimited. Users can configure this value based on their usecases. To avoid any data loss, this value should be more than the maximum retention period of any topic enabled with tiered storage in the cluster.
remote.log.metadata.manager.listener.name	Listener name to be used to connect to the local broker by RemoteLogMetadataManager implementation on the broker. This is a mandatory config while using the default RLMM implementation which is `org.apache.kafka.server.log.remote.metadata.storage.TopicBasedRemoteLogMetadataManager`. Respective endpoint address is passed with "bootstrap.servers" property while invoking RemoteLogMetadataManager#configure(Map<String, ?> props). This is used by kafka clients created in RemoteLogMetadataManager implementation.

*remote.log.metadata.**	Default RLMM implementation creates producer and consumer instances. Common client properties can be configured with `remote.log.metadata.common.client.` prefix. User can also pass properties specific to producer/consumer with `remote.log.metadata.producer.` and `remote.log.metadata.consumer.` prefixes. These will override properties with `remote.log.metadata.common.client.` prefix. Any other properties should be prefixed with the config: "remote.log.metadata.manager.impl.prefix", default value is "rlmm.config.". These configs will be passed to RemoteLogMetadataManager#configure(Map<String, ?> props). For example: "rlmm.config.remote.log.metadata.producer.batch.size=100" will set the `batch.size` config for the producer inside default RLMM.
remote.partition.remover.task.interval.ms	The interval at which remote partition remover runs to delete the remote storage of the partitions marked for deletion. Default value: 3600000 (1 hr)

Committed offsets file format

Committed offsets are stored in a local file `*_rlmm_committed_offsets*` under log dir. This file contains offset entry for each subscribed remote log metadata partition as "<partition-no> <offset>".

_rlmm_committed_offsets

```
0 2022
4 104
2 498
```

Internal flat-file store format of remote log metadata

RLMM stores the remote log metadata messages and builds materialized instances in a flat-file store for each user topic partition.

flat_file_format

```
<magic><topic-name><topic-id><metadata-topic-offset><sequence-of-serialized-entries>

magic:
    unsigned var int, version of this file format.
topic-name:
    string, topic name.
topic-id:
    uuid, uuid of topic
metadata-topic-offset:
    var long, offset of the remote log metadata topic partition upto which this topic partition's remote log
metadata is fetched.
serialized-entries:
   sequence of serialized entries defined as below, more types can be added later if needed.

Serialization of entry is done as mentioned below. This is very similar to the message format mentioned earlier
for storing into the metadata topic.

length    : unsigned var int, length of this entry which is sum of sizes of type, version, and data.
type      : unsigned var int, represents the value type. This value is 'apikey' as mentioned in the schema.
version   : unsigned var int, the 'version' number of the type as mentioned in the schema.
data      : record payload in kafka protocol message format, the schema is given below.

Both type and version are added before the data is serialized into record value.  Schema can be evolved by
adding a new version with the respective changes. A new type can also be supported by adding the respective
type and its version.


{
  "apiKey": 0,
  "type": "data",
  "name": "RemoteLogSegmentMetadataRecordStored",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    {
      "name": "SegmentId",
```

```
      "type": "uuid",
      "versions": "0+",
      "about": "Unique identifier of the log segment"
    },
    {
      "name": "StartOffset",
      "type": "int64",
      "versions": "0+",
      "about": "Start offset  of the segment."
    },
    {
      "name": "EndOffset",
      "type": "int64",
      "versions": "0+",
      "about": "End offset  of the segment."
    },
    {
      "name": "LeaderEpoch",
      "type": "int32",
      "versions": "0+",
      "about": "Leader epoch from which this segment instance is created or updated"
    },
    {
      "name": "MaxTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Maximum timestamp with in this segment."
    },
    {
      "name": "EventTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Event timestamp of this segment."
    },
    {
      "name": "SegmentLeaderEpochs",
      "type": "[]SegmentLeaderEpochEntry",
      "versions": "0+",
      "about": "Event timestamp of this segment.",
      "fields": [
        {
          "name": "LeaderEpoch",
          "type": "int32",
          "versions": "0+",
          "about": "Leader epoch"
        },
        {
          "name": "Offset",
          "type": "int64",
          "versions": "0+",
          "about": "Start offset for the leader epoch"
        }
      ]
    },
    {
      "name": "SegmentSizeInBytes",
      "type": "int32",
      "versions": "0+",
      "about": "Segment size in bytes"
    },
    {
      "name": "RemoteLogSegmentState",
      "type": "int8",
      "versions": "0+",
      "about": "State of the remote log segment"
    }
  ]
}


{
```

```
  "apiKey": 1,
  "type": "data",
  "name": "DeletePartitionStateRecord",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    {
      "name": "Epoch",
      "type": "int32",
      "versions": "0+",
      "about": "Epoch (controller or leader) from which this event is created. DELETE_PARTITION_MARKED is sent
by the controller. DELETE_PARTITION_STARTED and DELETE_PARTITION_FINISHED are sent by remote log metadata topic
partition leader."
    },
    {
      "name": "EventTimestamp",
      "type": "int64",
      "versions": "0+",
      "about": "Event timestamp of this segment."
    },
    {
      "name": "RemotePartitionDeleteState",
      "type": "int8",
      "versions": "0+",
      "about": "Deletion state of the remote partition"
    }
  ]
}
```

Message Formatter for the internal topic

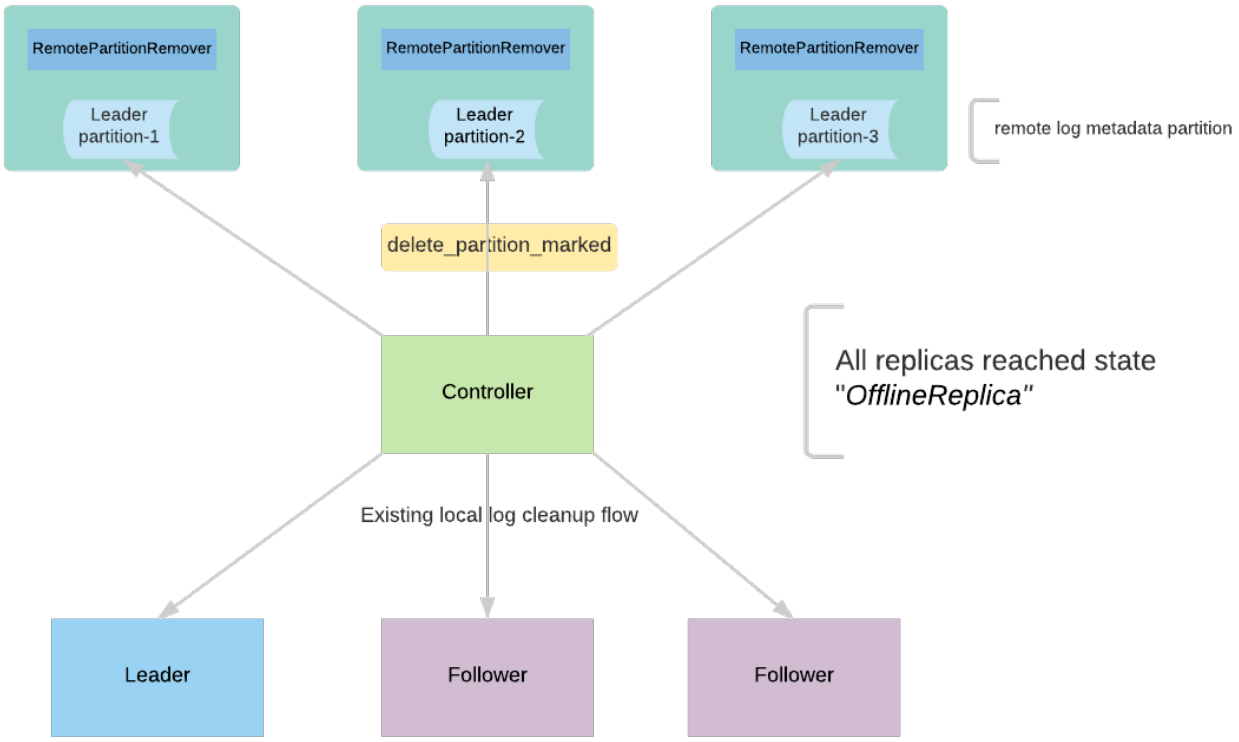
`org.apache.kafka.server.log.remote.storage.RemoteLogMetadataFormatter` can be used to format messages received from remote log metadata topic by console consumer. Users can pass properties mentioned in the below block with '–property' while running console consumer with this message formatter. The below block explains the format and it may change later. This formatter can be helpful for debugging purposes.

Internal message format

```
partition:<val><sep>message-offset:<val><sep>type:<RemoteLogSegmentMetadata | RemoteLogSegmentMetadataUpdate |
DeletePartitionState><sep>version:<_no_><vs>event-value:<string representation of the event>

val: represents the respective value of the key.
sep: represents the separator, default value is: ","

partition : Remote log metata topic partition number. This is optional.
Use print.partition property to print it, default is false

message-offset : Offset of this message in remote log metadata topic. This is optional.
Use print.message.offset property to print it, default is false

type: Event value type, which can be one of RemoteLogSegmentMetadata, RemoteLogSegmentMetadataUpdate,
DeletePartitionState values.

version: Version number of the event value type. This is optional.
Use print.version property to print it, default is false

Use print.all.event.value.fields to print the string representation of the event which will include all the
fields in the data, default property value is false.

Event value can be of any of the types below:

remote-log-segment-id is represented as "{id:<><sep>topicId:<val><sep>topicName:<val><sep>partition:<val>}" in
the event value.
topic-id-partition is represented as "{topicId:<val><sep>topicName:<val><sep>partition:<val>}" in the event
value.

For RemoteLogSegmentMetadata
default representation is "{remote-log-segment-id:<val><sep>start-offset:<val><sep>end-offset:<val><sep>leader-
epoch:<val><sep>remote-log-segment-state:<COPY_SEGMENT_STARTED | COPY_SEGMENT_FINISHED | DELETE_SEGMENT_STARTED
| DELETE_SEGMENT_FINISHED>}"

For RemoteLogSegmentMetadataUpdate
default representation is "{remote-log-segment-id:<val><sep>leader-epoch:<val><sep>remote-log-segment-state:
<COPY_SEGMENT_STARTED | COPY_SEGMENT_FINISHED | DELETE_SEGMENT_STARTED | DELETE_SEGMENT_FINISHED>}"

For DeletePartitionState
default representation is "{topic-id-partition:<val><sep>epoch:<val><sep>remote-partition-delete-state:
<DELETE_PARTITION_MARKED | DELETE_PARTITION_STARTED | DELETE_PARTITION_FINISHED>
```

Topic deletion lifecycle

The controller receives a delete request for a topic. It goes through the existing protocol of deletion and it makes all the replicas offline stop taking any fetch requests. After all the replicas reach the offline state, the controller publishes an event to the RemoteLogMetadataManager(RLMM) by marking the topic as deleted using RemoteLogMetadataManager.updateRemotePartitionDeleteMetadata with the state as RemotePartitionDeleteState#DELETE_PARTITION_MARKED. With KIP-516, topics are represented with uuid, and topics can be deleted asynchronously. This allows the remote logs can be garbage collected later by publishing the deletion marker into the remote log metadata topic. RLMM is responsible for asynchronously deleting all the remote log segments of a partition after receiving RemotePartitionDeleteState as DELETE_PARTITION_MARKED.

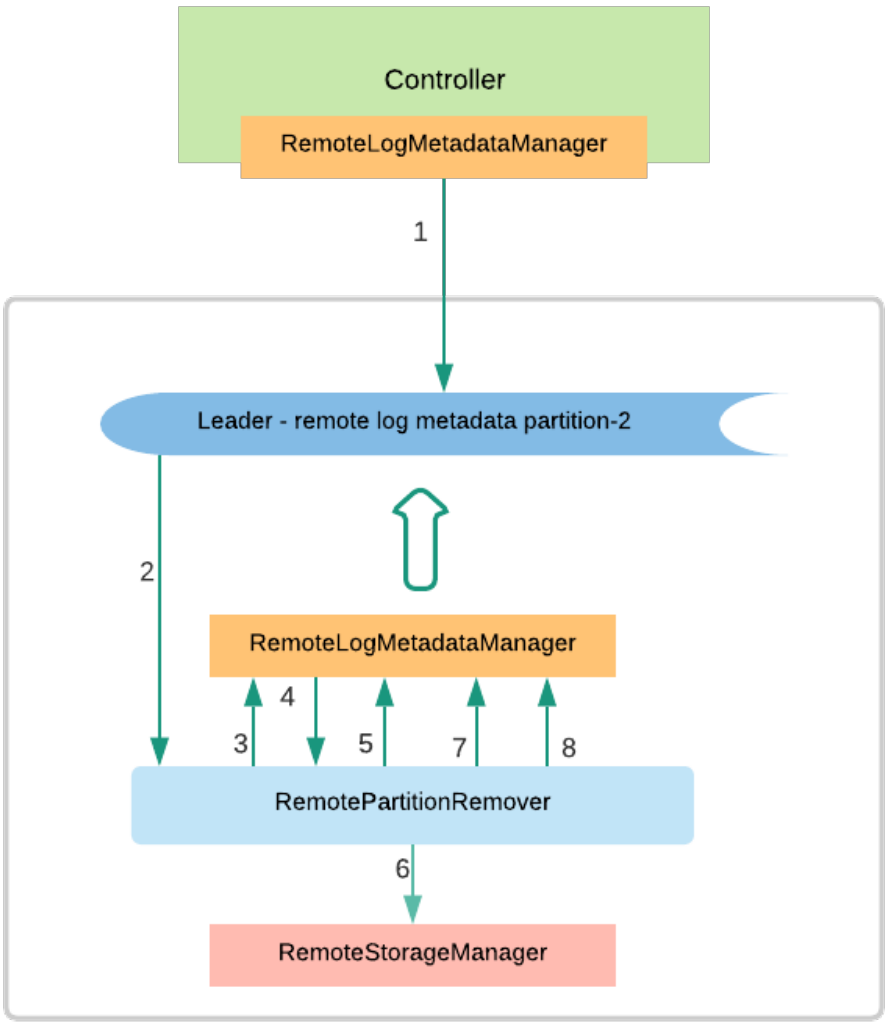


Default RLMM handles the remote partition deletion by using RemotePartitionRemover(RPRM).

RPRM instance is created on a broker with the leaders of the remote log segment metadata topic partitions. This task is responsible for removing remote storage of the topics marked for deletion. It consumes messages from those partitions remote log metadata partitions and filters the delete partition events which need to be processed. It collects those partitions and executes deletion of the respective segments using RemoteStorageManager. This is done at regular intervals of *remote.partition.remover.task.interval.ms* (default value of 1hr). It commits the processed offsets of metadata partitions once the deletions are executed successfully. This will also be helpful to handle leader failovers to a different replica so that it can start processing the messages where it left off.

RemotePartitionRemover(RPRM) processes the request with the following flow as mentioned in the below diagram.

1. The controller publishes DELETE_PARTITION_MARKED event to say that the partition is marked for deletion. There can be multiple events published when the controller restarts or failover and this event will be deduplicated by RPRM.
2. RPRM receives the DELETE_PARTITION_MARKED and processes it if it is not yet processed earlier.
3. RPRM publishes an event DELETE_PARTITION_STARTED that indicates the partition deletion has already been started.
4. RPRM gets all the remote log segments for the partition using RLMM and each of these remote log segments is deleted with the next steps.RLMM subscribes to the local remote log metadata partitions and it will have the segment metadata of all the user topic partitions associated with that remote log metadata partition.
5. Publish DELETE_SEGMENT_STARTED event with the segment id.
6. RPRM deletes the segment using RSM
7. Publish DELETE_SEGMENT_FINISHED event with segment id once it is successful.
8. Publish DELETE_PARTITION_FINISHED once all the segments have been deleted successfully.



Protocol Changes

ListOffsets

Currently, it supports the listing of offsets based on the earliest timestamp and the latest timestamp of the complete log. There is no change in the protocol but the new versions will start supporting listing earliest offsets based on the local logs but not only on the complete log including remote log. This protocol will be updated with the changes from KIP-516 but there are no changes required as mentioned earlier. Request and response versions will be bumped to version 7.

Fetch

We are bumping up fetch protocol to handle new error codes, there are no changes in request and response schemas. When a follower tries to fetch records for an offset that does not exist locally then it returns a new error `OFFSET_MOVED_TO_TIERED_STORAGE`. This is explained in detail here.

OFFSET_MOVED_TO_TIERED_STORAGE - when the requested offset is not available in local storage but it is moved to tiered storage.

Public Interfaces

Compacted topics will not have remote storage support.

Configs

System-Wide	*remote.log.storage.system.enable* - Whether to enable tier storage functionality in a broker or not. Valid values are `true` or `false` and the default value is false. This property gives backward compatibility. When it is true broker starts all the services required for tiered storage. *remote.log.storage.manager.class.name* - This is mandatory if the *remote.log.storage.system.enable is set as true.* remote.log.metadata.manager.class.name(optional) - This is an optional property. If this is not configured, Kafka uses an inbuilt metadata manager backed by an internal topic.
RemoteStorageManager	(These configs are dependent on remote storage manager implementation) *remote.log.storage.**
RemoteLogMetadataManager	(These configs are dependent on remote log metadata manager implementation) remote.log.metadata.*
Remote log manager related configuration.	*remote.log.index.file.cache.total.size.mb* The total size of the space allocated to store index files fetched from remote storage in the local storage. Default value: 1024 *remote.log.manager.thread.pool.size* Remote log thread pool size, which is used in scheduling tasks to copy segments, and clean up remote log segments. Default value: 4 *remote.log.manager.task.interval.ms* The interval at which the remote log manager runs the scheduled tasks like copy segments, and clean up remote log segments. Default value: 30,000 Remote log manager tasks are retried with the exponential backoff algorithm mentioned here. *remote.log.manager.task.retry.backoff.ms* The amount of time in milliseconds to wait before attempting the initial retry of a failed remote storage request. Default value: 500 *remote.log.manager.task.retry.backoff.max.ms* The maximum amount of time in milliseconds to wait before attempting to retry a failed remote storage request. Default value: 30,000 remote.log.manager.task.retry.jitter Random jitter amount applied to the `remote.log.manager.task.retry.backoff.ms` for computing the resultant backoff interval. This will avoid reconnection storms. Default value: 0.2 *remote.log.reader.threads* Remote log reader thread pool size, which is used in scheduling tasks to fetch data from remote storage. Default value: 5 *remote.log.reader.max.pending.tasks* Maximum remote log reader thread pool task queue size. If the task queue is full, broker will stop reading remote log segments. Default value: 100
Per Topic Configuration	Users can set the desired config for remote.storage.enable property for a topic, the default value is false. To enable tier storage for a topic, set remote.storage.enable as true. You can not disable this config once it is enabled. We will provide this feature in future versions. Below retention configs are similar to the log retention. This configuration is used to determine how long the log segments are to be retained in the local storage. Existing retention.* are retention configs for the topic partition which includes both local and remote storage. *local.retention.ms* The number of milli seconds to keep the local log segment before it gets deleted. If not set, the value in `log.retention.ms` is used. The effective value should always be less than or equal to retention.ms value. *local.retention.bytes* The maximum size of local log segments that can grow for a partition before it deletes the old segments. If not set, the value in `log.retention.bytes` is used. The effective value should always be less than or equal to retention.bytes value.

Remote Storage Manager

`RemoteStorageManager` is an interface to provide the lifecycle of remote log segments and indexes. More details about how we arrived at this interface are discussed in the document. We will provide a simple implementation of RSM to get a better understanding of the APIs. HDFS and S3 implementation are planned to be hosted in external repos and these will not be part of Apache Kafka repo. This is in line with the approach taken for Kafka connectors.

Copying and Deleting APIs are expected to be idempotent, so plugin implementations can retry safely and overwrite any partially copied content, or not failing when content is already deleted.

RemoteStorageManager

```
package org.apache.kafka.server.log.remote.storage;
...
/**
 * RemoteStorageManager provides the lifecycle of remote log segments that includes copy, fetch, and delete
from remote
 * storage.
 * <p>
 * Each upload or copy of a segment is initiated with {@link RemoteLogSegmentMetadata} containing {@link
RemoteLogSegmentId}
 * which is universally unique even for the same topic partition and offsets.
 * <p>
 * RemoteLogSegmentMetadata is stored in {@link RemoteLogMetadataManager} before and after copy/delete
operations on
 * RemoteStorageManager with the respective {@link RemoteLogSegmentState}. {@link RemoteLogMetadataManager} is
 * responsible for storing and fetching metadata about the remote log segments in a strongly consistent manner.
 * This allows RemoteStorageManager to store segments even in eventually consistent manner as the metadata is
already
 * stored in a consistent store.
 * <p>
 * All these APIs are still evolving.
 */
@InterfaceStability.Unstable
public interface RemoteStorageManager extends Configurable, Closeable {


    /**
     * Type of the index file.
     */
    enum IndexType {
        /**
         * Represents offset index.
         */
        Offset,

        /**
         * Represents timestamp index.
         */
        Timestamp,

        /**
         * Represents producer snapshot index.
         */
        ProducerSnapshot,

        /**
         * Represents transaction index.
         */
        Transaction,

        /**
         * Represents leader epoch index.
         */
        LeaderEpoch,
    }

    /**
     * Copies the given {@link LogSegmentData} provided for the given {@code remoteLogSegmentMetadata}. This
includes
     * log segment and its auxiliary indexes like offset index, time index, transaction index, leader epoch
index, and
     * producer snapshot index.
     * <p>
     * Invoker of this API should always send a unique id as part of {@link
RemoteLogSegmentMetadata#remoteLogSegmentId()}
     * even when it retries to invoke this method for the same log segment data.
     * <p>
     * This operation is expected to be idempotent. If a copy operation is retried and there is existing
content already written,
```

```
     * it should be overwritten, and do not throw {@link RemoteStorageException}
     *
     * @param remoteLogSegmentMetadata metadata about the remote log segment.
     * @param logSegmentData           data to be copied to tiered storage.
     * @throws RemoteStorageException if there are any errors in storing the data of the segment.
     */
    void copyLogSegmentData(RemoteLogSegmentMetadata remoteLogSegmentMetadata,
                            LogSegmentData logSegmentData)
            throws RemoteStorageException;

    /**
     * Returns the remote log segment data file/object as InputStream for the given {@link
RemoteLogSegmentMetadata}
     * starting from the given startPosition. The stream will end at the end of the remote log segment data file
/object.
     *
     * @param remoteLogSegmentMetadata metadata about the remote log segment.
     * @param startPosition            start position of log segment to be read, inclusive.
     * @return input stream of the requested log segment data.
     * @throws RemoteStorageException          if there are any errors while fetching the desired segment.
     * @throws RemoteResourceNotFoundException the requested log segment is not found in the remote storage.
     */
    InputStream fetchLogSegment(RemoteLogSegmentMetadata remoteLogSegmentMetadata,
                                int startPosition) throws RemoteStorageException;

    /**
     * Returns the remote log segment data file/object as InputStream for the given {@link
RemoteLogSegmentMetadata}
     * starting from the given startPosition. The stream will end at the smaller of endPosition and the end of
the
     * remote log segment data file/object.
     *
     * @param remoteLogSegmentMetadata metadata about the remote log segment.
     * @param startPosition            start position of log segment to be read, inclusive.
     * @param endPosition              end position of log segment to be read, inclusive.
     * @return input stream of the requested log segment data.
     * @throws RemoteStorageException          if there are any errors while fetching the desired segment.
     * @throws RemoteResourceNotFoundException the requested log segment is not found in the remote storage.
     */
    InputStream fetchLogSegment(RemoteLogSegmentMetadata remoteLogSegmentMetadata,
                                int startPosition,
                                int endPosition) throws RemoteStorageException;

    /**
     * Returns the index for the respective log segment of {@link RemoteLogSegmentMetadata}.
     * <p>
     * If the index is not present (e.g. Transaction index may not exist because segments create prior to
     * version 2.8.0 will not have transaction index associated with them.),
     * throws {@link RemoteResourceNotFoundException}
     *
     * @param remoteLogSegmentMetadata metadata about the remote log segment.
     * @param indexType                type of the index to be fetched for the segment.
     * @return input stream of the requested index.
     * @throws RemoteStorageException          if there are any errors while fetching the index.
     * @throws RemoteResourceNotFoundException the requested index is not found in the remote storage.
     * The caller of this function are encouraged to re-create the indexes from the segment
     * as the suggested way of handling this error.
     */
    InputStream fetchIndex(RemoteLogSegmentMetadata remoteLogSegmentMetadata,
                           IndexType indexType) throws RemoteStorageException;

    /**
     * Deletes the resources associated with the given {@code remoteLogSegmentMetadata}. Deletion is considered
as
     * successful if this call returns successfully without any errors. It will throw {@link
RemoteStorageException} if
     * there are any errors in deleting the file.
     * <p>
     * This operation is expected to be idempotent. If resources are not found, it is not expected to
     * throw {@link RemoteResourceNotFoundException} as it may be already removed from a previous attempt.
     *
```

```
     * @param remoteLogSegmentMetadata metadata about the remote log segment to be deleted.
     * @throws RemoteStorageException          if there are any storage related errors occurred.
     */
    void deleteLogSegmentData(RemoteLogSegmentMetadata remoteLogSegmentMetadata) throws RemoteStorageException;
  }


package org.apache.kafka.common;
...
public class TopicIdPartition {

    private final UUID topicId;
    private final TopicPartition topicPartition;

    public TopicIdPartition(UUID topicId, TopicPartition topicPartition) {
        Objects.requireNonNull(topicId, "topicId can not be null");
        Objects.requireNonNull(topicPartition, "topicPartition can not be null");
        this.topicId = topicId;
        this.topicPartition = topicPartition;
    }

    public UUID topicId() {
        return topicId;
    }

    public TopicPartition topicPartition() {
        return topicPartition;
    }

...
}


package org.apache.kafka.server.log.remote.storage;
...
/**
 * This represents a universally unique identifier associated to a topic partition's log segment. This will be
 * regenerated for every attempt of copying a specific log segment in {@link RemoteStorageManager#copyLogSegment
(RemoteLogSegmentMetadata, LogSegmentData)}.
 */
public class RemoteLogSegmentId implements Comparable<RemoteLogSegmentId>, Serializable {
    private static final long serialVersionUID = 1L;

    private final TopicIdPartition topicIdPartition;
    private final UUID id;

    public RemoteLogSegmentId(TopicIdPartition topicIdPartition, UUID id) {
        this.topicIdPartition = requireNonNull(topicIdPartition);
        this.id = requireNonNull(id);
    }

    /**
     * Returns TopicIdPartition of this remote log segment.
     *
     * @return
     */
    public TopicIdPartition topicIdPartition() {
        return topicIdPartition;
    }

    /**
     * Returns Universally Unique Id of this remote log segment.
     *
     * @return
     */
    public UUID id() {
        return id;
    }
...
}
```

```
package org.apache.kafka.server.log.remote.storage;
...
/**
 * It describes the metadata about the log segment in the remote storage.
 */
public class RemoteLogSegmentMetadata implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
     * Universally unique remote log segment id.
     */
    private final RemoteLogSegmentId remoteLogSegmentId;

    /**
     * Start offset of this segment.
     */
    private final long startOffset;

    /**
     * End offset of this segment.
     */
    private final long endOffset;

    /**
     * Leader epoch of the broker.
     */
    private final int leaderEpoch;

    /**
     * Maximum timestamp in the segment
     */
    private final long maxTimestamp;

    /**
     * Epoch time at which the respective {@link #state} is set.
     */
    private final long eventTimestamp;

    /**
     * LeaderEpoch vs offset for messages with in this segment.
     */
    private final Map<Int, Long> segmentLeaderEpochs;

    /**
     * Size of the segment in bytes.
     */
    private final int segmentSizeInBytes;

    /**
     * It indicates the state in which the action is executed on this segment.
     */
    private final RemoteLogSegmentState state;

    /**
     * @param remoteLogSegmentId  Universally unique remote log segment id.
     * @param startOffset         Start offset of this segment.
     * @param endOffset           End offset of this segment.
     * @param maxTimestamp        Maximum timestamp in this segment
     * @param leaderEpoch         Leader epoch of the broker.
     * @param eventTimestamp      Epoch time at which the remote log segment is copied to the remote tier
storage.
     * @param segmentSizeInBytes  Size of this segment in bytes.
     * @param state               State of the respective segment of remoteLogSegmentId.
     * @param segmentLeaderEpochs leader epochs occurred with in this segment
     */
    public RemoteLogSegmentMetadata(RemoteLogSegmentId remoteLogSegmentId, long startOffset, long endOffset,
                                    long maxTimestamp, int leaderEpoch, long eventTimestamp,
                                    int segmentSizeInBytes, RemoteLogSegmentState state, Map<Int, Long>
segmentLeaderEpochs) {
        this.remoteLogSegmentId = remoteLogSegmentId;
```

```
        this.startOffset = startOffset;
        this.endOffset = endOffset;
        this.leaderEpoch = leaderEpoch;
        this.maxTimestamp = maxTimestamp;
        this.eventTimestamp = eventTimestamp;
        this.segmentLeaderEpochs = segmentLeaderEpochs;
        this.state = state;
        this.segmentSizeInBytes = segmentSizeInBytes;
    }

    /**
     * @return unique id of this segment.
     */
    public RemoteLogSegmentId remoteLogSegmentId() {
        return remoteLogSegmentId;
    }

    /**
     * @return Start offset of this segment(inclusive).
     */
    public long startOffset() {
        return startOffset;
    }

    /**
     * @return End offset of this segment(inclusive).
     */
    public long endOffset() {
        return endOffset;
    }

    /**
     * @return Leader or controller epoch of the broker from where this event occurred.
     */
    public int brokerEpoch() {
        return brokerEpoch;
    }

    /**
     * @return Epoch time at which this evcent is occurred.
     */
    public long eventTimestamp() {
        return eventTimestamp;
    }

    /**
     * @return
     */
    public int segmentSizeInBytes() {
        return segmentSizeInBytes;
    }

    public RemoteLogSegmentState state() {
        return state;
    }

    public long maxTimestamp() {
        return maxTimestamp;
    }

    public Map<Int, Long> segmentLeaderEpochs() {
        return segmentLeaderEpochs;
    }

...
}

package org.apache.kafka.server.log.remote.storage;
...
public class LogSegmentData {
```

```
    private final File logSegment;
    private final File offsetIndex;
    private final File timeIndex;
    private final File txnIndex;
    private final File producerIdSnapshotIndex;
    private final ByteBuffer leaderEpochIndex;

    public LogSegmentData(File logSegment, File offsetIndex, File timeIndex, File txnIndex, File
producerIdSnapshotIndex,
                          ByteBuffer leaderEpochIndex) {
        this.logSegment = logSegment;
        this.offsetIndex = offsetIndex;
        this.timeIndex = timeIndex;
        this.txnIndex = txnIndex;
        this.producerIdSnapshotIndex = producerIdSnapshotIndex;
        this.leaderEpochIndex = leaderEpochIndex;
    }

    public File logSegment() {
        return logSegment;
    }

    public File offsetIndex() {
        return offsetIndex;
    }

    public File timeIndex() {
        return timeIndex;
    }

    public File txnIndex() {
        return txnIndex;
    }

    public File producerIdSnapshotIndex() {
        return producerIdSnapshotIndex;
    }

    public ByteBuffer leaderEpochIndex() {
        return leaderEpochIndex;
    }
...
}
```

RemoteLogMetadataManager

`RemoteLogMetadataManager` is an interface to provide the lifecycle of metadata about remote log segments with strongly consistent semantics. There is a default implementation that uses an internal topic. Users can plugin their own implementation if they intend to use another system to store remote log segment metadata.

RemoteLogMetadataManager

```
package org.apache.kafka.server.log.remote.storage;
...
/**
 * This interface provides storing and fetching remote log segment metadata with strongly consistent semantics.
 * <p>
 * This class can be plugged in to Kafka cluster by adding the implementation class as
 * <code>remote.log.metadata.manager.class.name</code> property value. There is an inbuilt implementation
backed by
 * topic storage in the local cluster. This is used as the default implementation if
 * remote.log.metadata.manager.class.name is not configured.
 * </p>
 * <p>
 * <code>remote.log.metadata.manager.class.path</code> property is about the class path of the
RemoteLogStorageManager
 * implementation. If specified, the RemoteLogStorageManager implementation and its dependent libraries will be
```

```
loaded
 * by a dedicated classloader which searches this class path before the Kafka broker class path. The syntax of
this
 * parameter is same with the standard Java class path string.
 * </p>
 * <p>
 * <code>remote.log.metadata.manager.listener.name</code> property is about listener name of the local broker
to which
 * it should get connected if needed by RemoteLogMetadataManager implementation. When this is configured all
other
 * required properties can be passed as properties with prefix of 'remote.log.metadata.manager.listener.
 * </p>
 * "cluster.id", "broker.id" and all other properties prefixed with "remote.log.metadata." are passed when
 * {@link #configure(Map)} is invoked on this instance.
 * <p>
 */
@InterfaceStability.Evolving
public interface RemoteLogMetadataManager extends Configurable, Closeable {

    /**
     * Asynchronously adds {@link RemoteLogSegmentMetadata} with the containing {@link RemoteLogSegmentId} into
{@link RemoteLogMetadataManager}.
     * <p>
     * RemoteLogSegmentMetadata is identified by RemoteLogSegmentId and it should have the initial state which
is {@link RemoteLogSegmentState#COPY_SEGMENT_STARTED}.
     * <p>
     * {@link #updateRemoteLogSegmentMetadata(RemoteLogSegmentMetadataUpdate)} should be used to update an
existing RemoteLogSegmentMetadata.
     *
     * @param remoteLogSegmentMetadata metadata about the remote log segment.
     * @throws RemoteStorageException   if there are any storage related errors occurred.
     * @throws IllegalArgumentException if the given metadata instance does not have the state as {@link
RemoteLogSegmentState#COPY_SEGMENT_STARTED}
     * @return a Future which will complete once this operation is finished.
     */
    Future<Void> addRemoteLogSegmentMetadata(RemoteLogSegmentMetadata remoteLogSegmentMetadata) throws
RemoteStorageException;

    /**
     * This method is used to update the {@link RemoteLogSegmentMetadata} asynchronously. Currently, it allows
to update with the new
     * state based on the life cycle of the segment. It can go through the below state transitions.
     * <p>
     * <pre>
     * +----------------------+            +----------------------+
     * |COPY_SEGMENT_STARTED |----------->|COPY_SEGMENT_FINISHED |
     * +-------------------+-+            +--+-------------------+
     *                     |                 |
     *                     |                 |
     *                     v                 v
     *                  +--+-----------------+-+
     *                  |DELETE_SEGMENT_STARTED|
     *                  +-----------+----------+
     *                              |
     *                              |
     *                              v
     *                  +-----------+-----------+
     *                  |DELETE_SEGMENT_FINISHED|
     *                  +-----------------------+
     * </pre>
     * <p>
     * {@link RemoteLogSegmentState#COPY_SEGMENT_STARTED} - This state indicates that the segment copying to
remote storage is started but not yet finished.
     * {@link RemoteLogSegmentState#COPY_SEGMENT_FINISHED} - This state indicates that the segment copying to
remote storage is finished.
     * <br>
     * The leader broker copies the log segments to the remote storage and puts the remote log segment metadata
with the
     * state as “COPY_SEGMENT_STARTED” and updates the state as “COPY_SEGMENT_FINISHED” once the copy is
successful.
     * <p></p>
```

```
     * {@link RemoteLogSegmentState#DELETE_SEGMENT_STARTED} - This state indicates that the segment deletion is
started but not yet finished.
     * {@link RemoteLogSegmentState#DELETE_SEGMENT_FINISHED} - This state indicates that the segment is deleted
successfully.
     * <br>
     * Leader partitions publish both the above delete segment events when remote log retention is reached for
the
     * respective segments. Remote Partition Removers also publish these events when a segment is deleted as
part of
     * the remote partition deletion.
     *
     * @param remoteLogSegmentMetadataUpdate update of the remote log segment metadata.
     * @throws RemoteStorageException          if there are any storage related errors occurred.
     * @throws RemoteResourceNotFoundException when there are no resources associated with the given
remoteLogSegmentMetadataUpdate.
     * @throws IllegalArgumentException        if the given metadata instance has the state as {@link
RemoteLogSegmentState#COPY_SEGMENT_STARTED}
     * @return a Future which will complete once this operation is finished.
     */
    Future<Void> updateRemoteLogSegmentMetadata(RemoteLogSegmentMetadataUpdate remoteLogSegmentMetadataUpdate)
            throws RemoteStorageException;

    /**
     * Returns {@link RemoteLogSegmentMetadata} if it exists for the given topic partition containing the
offset with
     * the given leader-epoch for the offset, else returns {@link Optional#empty()}.
     *
     * @param topicIdPartition topic partition
     * @param epochForOffset   leader epoch for the given offset
     * @param offset           offset
     * @return the requested remote log segment metadata if it exists.
     * @throws RemoteStorageException if there are any storage related errors occurred.
     */
    Optional<RemoteLogSegmentMetadata> remoteLogSegmentMetadata(TopicIdPartition topicIdPartition,
                                                                int epochForOffset,
                                                                long offset)
            throws RemoteStorageException;

    /**
     * Returns the highest log offset of topic partition for the given leader epoch in remote storage. This is
used by
     * remote log management subsystem to know up to which offset the segments have been copied to remote
storage for
     * a given leader epoch.
     *
     * @param topicIdPartition topic partition
     * @param leaderEpoch      leader epoch
     * @return the requested highest log offset if exists.
     * @throws RemoteStorageException if there are any storage related errors occurred.
     */
    Optional<Long> highestOffsetForEpoch(TopicIdPartition topicIdPartition,
                                         int leaderEpoch) throws RemoteStorageException;

    /**
     * This method is used to update the metadata about remote partition delete event asynchronously.
Currently, it allows updating the
     * state ({@link RemotePartitionDeleteState}) of a topic partition in remote metadata storage. Controller
invokes
     * this method with {@link RemotePartitionDeleteMetadata} having state as {@link
RemotePartitionDeleteState#DELETE_PARTITION_MARKED}.
     * So, remote partition removers can act on this event to clean the respective remote log segments of the
partition.
     * <p><br>
     * In the case of default RLMM implementation, remote partition remover processes {@link
RemotePartitionDeleteState#DELETE_PARTITION_MARKED}
     * <ul>
     * <li> sends an event with state as {@link RemotePartitionDeleteState#DELETE_PARTITION_STARTED}
     * <li> gets all the remote log segments and deletes them.
     * <li> sends an event with state as {@link RemotePartitionDeleteState#DELETE_PARTITION_FINISHED} once all
the remote log segments are
     * deleted.
```

```
     * </ul>
     *
     * @param remotePartitionDeleteMetadata update on delete state of a partition.
     * @throws RemoteStorageException          if there are any storage related errors occurred.
     * @throws RemoteResourceNotFoundException when there are no resources associated with the given
remotePartitionDeleteMetadata.
     * @return a Future which will complete once this operation is finished.
     */
    Future<Void> putRemotePartitionDeleteMetadata(RemotePartitionDeleteMetadata remotePartitionDeleteMetadata)
            throws RemoteStorageException;

    /**
     * Returns all the remote log segment metadata of the given topicIdPartition.
     * <p>
     * Remote Partition Removers uses this method to fetch all the segments for a given topic partition, so
that they
     * can delete them.
     *
     * @return Iterator of all the remote log segment metadata for the given topic partition.
     */
    Iterator<RemoteLogSegmentMetadata> listRemoteLogSegments(TopicIdPartition topicIdPartition)
            throws RemoteStorageException;

    /**
     * Returns iterator of remote log segment metadata, sorted by {@link RemoteLogSegmentMetadata#startOffset()}
in
     * ascending order which contains the given leader epoch. This is used by remote log retention management
subsystem
     * to fetch the segment metadata for a given leader epoch.
     *
     * @param topicIdPartition topic partition
     * @param leaderEpoch      leader epoch
     * @return Iterator of remote segments, sorted by start offset in ascending order.
     */
    Iterator<RemoteLogSegmentMetadata> listRemoteLogSegments(TopicIdPartition topicIdPartition,
                                                             int leaderEpoch) throws RemoteStorageException;

    /**
     * This method is invoked only when there are changes in leadership of the topic partitions that this
broker is
     * responsible for.
     *
     * @param leaderPartitions   partitions that have become leaders on this broker.
     * @param followerPartitions partitions that have become followers on this broker.
     */
    void onPartitionLeadershipChanges(Set<TopicIdPartition> leaderPartitions,
                                      Set<TopicIdPartition> followerPartitions);

    /**
     * This method is invoked only when the topic partitions are stopped on this broker. This can happen when a
     * partition is emigrated to other broker or a partition is deleted.
     *
     * @param partitions topic partitions that have been stopped.
     */
    void onStopPartitions(Set<TopicIdPartition> partitions);
}
package org.apache.kafka.server.log.remote.storage;
...
/**
 * It describes the metadata about the log segment in the remote storage.
 */
public class RemoteLogSegmentMetadataUpdate implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
     * Universally unique remote log segment id.
     */
    private final RemoteLogSegmentId remoteLogSegmentId;

    /**
```

```
     * Epoch time at which the respective {@link #state} is set.
     */
    private final long eventTimestamp;

    /**
     * Leader epoch of the broker from where this event occurred.
     */
    private final int leaderEpoch;

    /**
     * It indicates the state in which the action is executed on this segment.
     */
    private final RemoteLogSegmentState state;

    /**
     * @param remoteLogSegmentId  Universally unique remote log segment id.
     * @param eventTimestamp      Epoch time at which the remote log segment is copied to the remote tier
storage.
     * @param leaderEpoch         Leader epoch of the broker from where this event occurred.
     * @param state               state of the remote log segment.
     */
    public RemoteLogSegmentMetadataUpdate(RemoteLogSegmentId remoteLogSegmentId,
                                          long eventTimestamp,
                                          int leaderEpoch,
                                          RemoteLogSegmentState state) {
        this.remoteLogSegmentId = remoteLogSegmentId;
        this.eventTimestamp = eventTimestamp;
        this.leaderEpoch = leaderEpoch;
        this.state = state;
    }

    public RemoteLogSegmentId remoteLogSegmentId() {
        return remoteLogSegmentId;
    }

    public long createdTimestamp() {
        return eventTimestamp;
    }

    public RemoteLogSegmentState state() {
        return state;
    }

    public int leaderEpoch() {
        return leaderEpoch;
    }
...
}

package org.apache.kafka.server.log.remote.storage;
...
public class RemotePartitionDeleteMetadata {

    private final TopicIdPartition topicPartition;
    private final RemotePartitionDeleteState state;
    private final long eventTimestamp;
    private final int epoch;

    public RemotePartitionDeleteMetadata(TopicIdPartition topicPartition, RemotePartitionDeleteState state,
long eventTimestamp, int epoch) {
        Objects.requireNonNull(topicPartition);
        Objects.requireNonNull(state);
        if(state != RemotePartitionDeleteState.DELETE_PARTITION_MARKED && state != RemotePartitionDeleteState.
DELETE_PARTITION_STARTED
                && state != RemotePartitionDeleteState.DELETE_PARTITION_FINISHED) {
            throw new IllegalArgumentException("state should be one of the delete partition states");
        }
        this.topicPartition = topicPartition;
        this.state = state;
        this.eventTimestamp = eventTimestamp;
        this.epoch = epoch;
```

```
    }

    public TopicIdPartition topicPartition() {
        return topicPartition;
    }

    public RemotePartitionDeleteState state() {
        return state;
    }

    public long eventTimestamp() {
        return eventTimestamp;
    }

    public int epoch() {
        return epoch;
    }

...
}


package org.apache.kafka.server.log.remote.storage;
...
/**
 * It indicates the deletion state of the remote topic partition. This will be based on the action executed on
this
 * partition by the remote log service implementation.
 * <p>
 */
public enum RemotePartitionDeleteState {

    /**
     * This is used when a topic/partition is deleted by controller.
     * This partition is marked for delete by controller. That means, all its remote log segments are eligible
for
     * deletion so that remote partition removers can start deleting them.
     */
    DELETE_PARTITION_MARKED((byte) 0),

    /**
     * This state indicates that the partition deletion is started but not yet finished.
     */
    DELETE_PARTITION_STARTED((byte) 1),

    /**
     * This state indicates that the partition is deleted successfully.
     */
    DELETE_PARTITION_FINISHED((byte) 2);

    private static final Map<Byte, RemotePartitionDeleteState> STATE_TYPES = Collections.unmodifiableMap(
            Arrays.stream(values()).collect(Collectors.toMap(RemotePartitionDeleteState::id, Function.
identity())));

    private final byte id;

    RemotePartitionDeleteState(byte id) {
        this.id = id;
    }

    public byte id() {
        return id;
    }

    public static RemotePartitionDeleteState forId(byte id) {
        return STATE_TYPES.get(id);
    }
...
}

package org.apache.kafka.server.log.remote.storage;
```

```
...
/**
 * It indicates the state of the remote log segment. This will be based on the action executed on this
 * segment by the remote log service implementation.
 * <p>
 */
public enum RemoteLogSegmentState {

    /**
     * This state indicates that the segment copying to remote storage is started but not yet finished.
     */
    COPY_SEGMENT_STARTED((byte) 0),

    /**
     * This state indicates that the segment copying to remote storage is finished.
     */
    COPY_SEGMENT_FINISHED((byte) 1),

    /**
     * This state indicates that the segment deletion is started but not yet finished.
     */
    DELETE_SEGMENT_STARTED((byte) 2),

    /**
     * This state indicates that the segment is deleted successfully.
     */
    DELETE_SEGMENT_FINISHED((byte) 3),

    private static final Map<Byte, RemoteLogSegmentState> STATE_TYPES = Collections.unmodifiableMap(
            Arrays.stream(values()).collect(Collectors.toMap(RemoteLogSegmentState::id, Function.identity())));

    private final byte id;

    RemoteLogSegmentState(byte id) {
        this.id = id;
    }

    public byte id() {
        return id;
    }

    public static RemoteLogSegmentState forId(byte id) {
        return STATE_TYPES.get(id);
    }
...
}
```

New Metrics

The following new metrics will be added:

MBean	description
kafka.server:type=BrokerTopicMetrics, name=RemoteReadRequestsPerSec, topic=([-.w]+)	Number of remote storage read requests per second.
kafka.server:type=BrokerTopicMetrics, name=RemoteBytesInPerSec, topic=([-.w]+)	Number of bytes read from remote storage per second.
kafka.server:type=BrokerTopicMetrics, name=RemoteReadErrorPerSec, topic=([-.w]+)	Number of remote storage read errors per second.
kafka.log.remote:type=RemoteStorageThreadPool, name=RemoteLogReaderTaskQueueSize	Number of remote storage read tasks pending for execution.
kafka.log.remote:type=RemoteStorageThreadPool, name=RemoteLogReaderAvgIdlePercent	Average idle percent of the remote storage reader thread pool.
kafka.log.remote:type=RemoteLogManager, name=RemoteLogManagerTasksAvgIdlePercent	Average idle percent of RemoteLogManager thread pool.

kafka.server:type=BrokerTopicMetrics, name=RemoteBytesOutPerSec, topic=([-.w]+)	Number of bytes copied to remote storage per second.
kafka.server:type=BrokerTopicMetrics, name=RemoteWriteErrorPerSec, topic=([-.w]+)	Number of remote storage write errors per second.

Some of these metrics have been updated with new names as part of KIP-930

Upgrade

Follow the steps mentioned in Kafka upgrade to reach the state where all brokers are running on the latest binaries with the respective "inter.broker.protocol" and "log.message.format" versions. Tiered storage requires the message format to be > 0.11.

To enable tiered storage subsytems, a rolling restart should be done by enabling "remote.log.storage.system.enable" on all brokers.

You can enable tiered storage by setting “remote.storage.enable” to true on the desired topics. Before enabling tiered storage, you should make sure the producer snapshots are built for all the segments for that topic in all followers. You should wait till the log retention occurs for all the segments so that all the segments have producer snapshots. Because follower replicas for topics with tier storage enabled, need the respective producer snapshot for each segment for reconciling the state as mentioned in the earlier follower fetch protocol section.

Downgrade

Downgrade to earlier versions(> 2.1) is possible but the data available only on remote storage will not be available. There will be a few files that are created in remote index cache directory($log.dir/remote-log-index-cache) and other remote log segment metadata cache files that need to be cleaned up by the user. We may provide a script to cleanup the cache files created by tiered storage.Users have to manually delete the data in remote storage based on the bucket or dir configured with tiered storage.

Limitations

- Once tier storage is enabled for a topic, it can not be disabled. We will add this feature in future versions. One possible workaround is to create a new topic and copy the data from the desired offset and delete the old topic. Another possible work around is to set the log.local.retention.ms same as retention.ms and wait until the local retention catches up until complete log retention. This will make the complete data available locally. After that, set remote.storage.enable as false to disable tiered storage on a topic.
- Multiple Log dirs on a broker are not supported (JBOD related features).
- Tiered storage is not supported for compacted topics.

Integration and System tests

For integration tests, we use file based(*LocalTieredStorage*) RemoteStorageManager(RSM) . For system tests, we plan to have a single node HDFS cluster in one of the containers and use HDFS RSM implementation.

Feature Test

Feature test cases and test results are documented in this google spreadsheet.

Performance Test Results

We have tested the performance of the initial implementation of this proposal.

The cluster configuration:

1. 5 brokers
2. 20 CPU cores, 256GB RAM (each broker)
3. 2TB * 22 hard disks in RAID0 (each broker)
4. Hardware RAID card with NV-memory write cache
5. 20Gbps network
6. snappy compression
7. 6300 topic-partitions with 3 replicas
8. remote storage uses HDFS

Each test case is tested under 2 types of workload (acks=all and acks=1)

	Workload-1 **(at-least-once, acks=all)**	**Workload-2** **(acks=1)**
Producers	10 producers 30MB / sec / broker (leader) ~62K messages / sec / broker (leader)	10 producers 55MB / sec / broker (leader) ~120K messages / sec / broker (leader)

In-sync Consumers	10 consumers 120MB / sec / broker ~250K messages / sec / broker	10 consumers 220MB / sec / broker ~480K messages / sec / broker

Test case 1 (Normal case):

Normal traffic as described above.

		with tiered storage	without tiered storage
Workload-1	Avg P99 produce latency	25ms	21ms
(acks=all, low traffic)	Avg P95 produce latency	14ms	13ms
Workload-2	Avg P99 produce latency	9ms	9ms
(acks=1, high traffic)	Avg P95 produce latency	4ms	4ms

We can see there is a little overhead when tiered storage is turned on. This is expected, as the brokers have to ship segments to remote storage, and sync the remote segment metadata between brokers. With at-least-once (acks=all) produce, the produce latency is slightly increased when tiered storage is turned on. With acks=1 produce, the produce latency is almost not changed when tiered storage is turned on.

Test case 2 (out-of-sync consumers catching up):

In addition to the normal traffic, 9 out-of-sync consumers consume 180MB/s per broker (or 900MB/s in total) old data.

With tiered storage, the old data is read from HDFS. Without tiered storage, the old data is read from local disk.

		with tiered storage	without tiered storage
Workload-1	Avg P99 produce latency	42ms	60ms
(acks=all, low traffic)	Avg P95 produce latency	18ms	30ms
Workload-2	Avg P99 produce latency	10ms	10ms
(acks=1, high traffic)	Avg P95 produce latency	5ms	4ms

Consuming old data has a significant performance impact to acks=all producers. Without tiered storage, the P99 produce latency is almost ~1.5 times. With tiered storage, the performance impact is relatively lower, because remote storage reading does not compete with the local hard disk bandwidth with produce requests.

Consuming old data has little impact to acks=1 producers.

Test case 3 (rebuild broker):

Under the normal traffic, stop a broker, remove all the local data, and rebuild it without replication throttling. This case simulates replacing a broken broker server.

		with tiered storage	without tiered storage
Workload-1	Max avg P99 produce latency	56ms	490ms
(acks=all,	Max avg P95 produce latency	23ms	290ms
12TB data per broker)	Duration	2min	230min
Workload-2	Max avg P99 produce latency	12ms	10ms
(acks=1,	Max avg P95 produce latency	6ms	5ms
34TB data per broker)	Duration	4min	520min

With tiered storage, the rebuilding broker only needs to fetch the latest data that has not been shipped to remote storage. Without tiered storage, the rebuilt broker has to fetch all the data that has not expired from the other brokers. With the same log retention time, tiered storage reduced the rebuilding time by more than 100 times.

Without tiered storage, the rebuilding broker has to read a large amount of data from the local hard disks of the leaders. This competes for page cache and local disk bandwidth with the normal traffic and dramatically increases the acks=all produce latency.

Future work

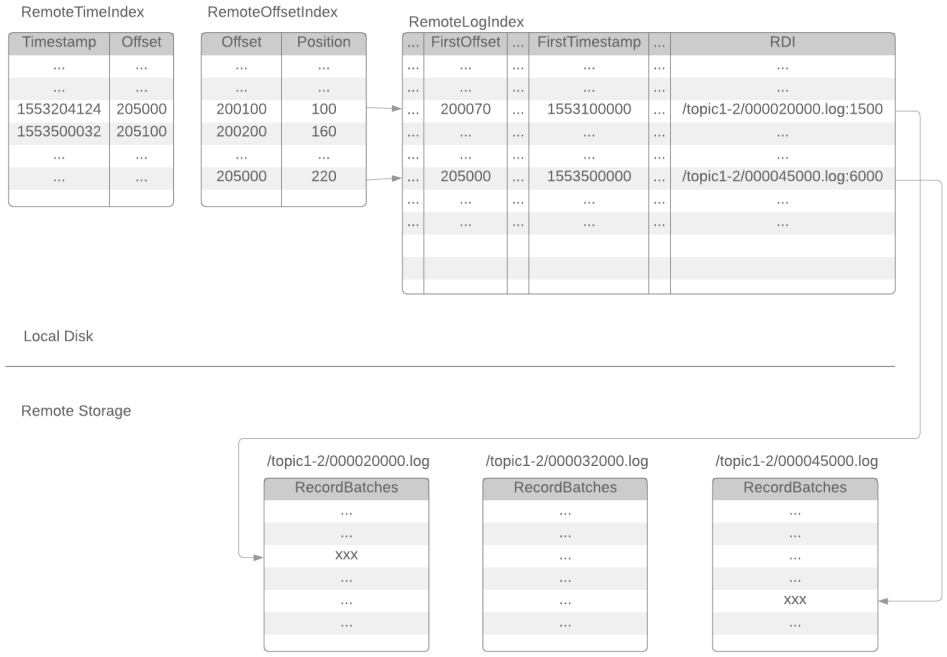
- Enhance RLMM local file-based cache with RocksDB to avoid loading the whole cache inmemory.

- Enhance RLMM implementation based on topic based storage pointing to a target Kafka cluster instead of using a system level topic within the cluster.
- Improve default RLMM implementation with a less chatty protocol.
- Support disabling tiered storage for a topic.
- Add a system level config to enable tiered storage for all the topics in a cluster.
- Recovery mechanism in case of the broker or cluster failure.
    - This is to be done by fetching the remote log metadata from RemoteStorageManager.
- Recovering from remote log metadata topic partitions truncation
- Extract RPMM as a separate task and allow any RLMM implementation to reuse the task for deletion of remote segments and complete the remote partition deletion.

Alternatives considered

Following alternatives were considered:

1. Replace all local storage with remote storage - Instead of using local storage on Kafka brokers, only remote storage is used for storing log segments and offset index files. While this has the benefits related to reducing the local storage, it has the problem of not leveraging the OS page cache and local disk for efficient latest reads as done in Kafka today.
2. Implement Kafka API on another store - This is an approach that is taken by some vendors where Kafka API is implemented on a different distributed, scalable storage (example HDFS). Such an option does not leverage Kafka other than API compliance and requires the much riskier option of replacing the entire Kafka cluster with another system.
3. Client directly reads remote log segments from the remote storage - The log segments on the remote storage can be directly read by the client instead of serving it from Kafka broker. This reduces Kafka broker changes and has the benefits of removing an extra hop. However, this bypasses Kafka security completely, increases Kafka client library complexity and footprint, causes compatibility issues to the existing Kafka client libraries, and hence is not considered.
4. Store all remote segment metadata in remote storage. This approach works with the storage systems that provide strong consistent metadata, such as HDFS, but does not work with S3 and GCS. Frequently calling LIST API on S3 or GCS also incurs huge costs. So, we choose to store metadata in a Kafka topic in the default implementation but allow users to use other methods with their own RLMM implementations.
5. Cache all remote log indexes in local storage. Store remote log segment information in local storage.



Meeting Notes

23 Mar 2021

(Notes by Kowshik)

- Discussion:
    - Discussed implementation of highestLogOffset and listAllRemoteLogSegments methods in KIP-405 PR: https://github.com/apache/kafka/pull/10218.
    - Discussed implementation of state transition validation checks in RemoteLogSegmentState and cases where the source state can still be null.
    - Discussed Log layer refactor and the plan to extract the recovery logic out of the Log layer in a separate PR.
- Follow-ups:

- Satish to look into review comments on https://github.com/apache/kafka/pull/10218. Jun/Kowshik to review the PR whenever it is ready again.
- Satish to raise PR addressing last batch of review comments on the interface PR: https://github.com/apache/kafka/pull/10173.
- Kowshik to continue working on recovery logic refactor and Log layer refactor.
- **(Done)** Kowshik to update the external facing Log layer refactor proposal doc with details about the recovery logic refactor: https://docs.google.com/document/d/1dQJL4MCwqQJSPmZkVmVzshFZKuFy_bCPtubav4wBfHQ/edit# .

09 Feb 2021

- Notes
  - Discussed the downgrade path, KIP will be updated with that.
  - Discussed the limitation of not allowing disable tiered storage on a topic.
  - All are agreed that KIP is ready for voting.

26 Jan 2021

- Notes
  - Discussed the latest review comments from the mail thread.
  - Manikumar will review and provide comments.

12 Jan 2021

- Notes
  - Satish discussed the edge cases around upgrade path with KIP-516 updates. Jun clarified on how topic-id is received after IBP is udpated on all brokers.
  - Jun suggested to update the KIP with more details on Remote Partition Remover.
  - RLMM flat file format was discussed and Jun asked to clarify the header section.
  - Kowshik and Jun will provide Log layer refactoring writeup.

15 Dec 2020

- Notes
  - Discussed producer snapshot fix missing in 2.7
  - Satish discussed memory growth due to RLMM cache and it looks to be practically negligible. The proposal is to use inmemory cache and checkpoint that to disk.
  - Satish will update the KIP with Upgrade path.
  - Kowshik and Jun will look into LOg refactoring.

08 Dec 2020

- Discussion Recording
- Notes

1. Tiered storage upgrade path dicussion:

   - Details need to be documented in the KIP.
   - Current upgrade path plan is based on IBP bump.
   - Enabling of the remote log components may not mean all topics are eligible for tiering at the same time.
   - Should tiered storage be enabled on all brokers before enabling it on any brokers?
   - Is there any replication path dependency for enabling tiered storage?

2. RLMM persistence format:

   - We agreed to document the persistence format for the materialized state of default RLMM implementation (topic-based).
   - (carry over from earlier discussion) For the file-based design, we don't know yet the % of increase in memory, assuming the majority of segments are in remote storage. It will be useful to document an estimation for this.

3. Topic deletion lifecycle discussion:

   - Under topic deletion lifecycle, step (4) it would be useful to mention how the RemotePartitionRemover (RPRM) gets the list of segments to be deleted, and whether it has any dependency with the RLMM topic.

4. Log layer discussion:

   - We discussed the complexities surrounding making code changes to Log layer (Log.scala).
   - Today, the Log class holds attributes and behavior related with local log. In the future, we would have to change the Log layer such that it would also contain the logic for the tiered portion of the log. This addition can pose a maintenance challenge.
   - Some of the existing attributes in the Log layer such as LeaderEpochCache and ProducerStateManager can be related with global view of the log too (i.e. global log is local log + tiered log). It can be therefore useful to think about preparatory refactoring, to see whether we can separate responsibilities related with the local log from the tiered log, and, perhaps provide a global view of the log that combines together both as and when required. The global view of the log could manage the lifecycle of LeaderEpochCache and ProducerStateManager.

Follow-ups:

- KIP-405 updates (upgrade path, RLMM file format and topic deletion)

- Log layer changes

(Notes taken by Kowshik)

01 Dec 2020

- Discussion Recording
- Notes
- Satish discussed KIP-405 updates:

1. Addressed some of the outstanding review comments from previous weeks.
2. Remote log manager (RLM) cache configuration was added.
3. Updated default values in the KIP for certain configuration parameters.
4. RLMM committed offsets are stored in separate files.
5. Initial version: go ahead with in-memory RLMM materializer implementation for now. Future switch to RocksDB seems feasible since it is an internal change only to RLMM cache.
6. Yet to update the KIP with KIP-516 (topic ID) changes.
7. Tiered storage upgrade path details are a work-in-progress. Will be added to the KIP.

RLMM cache choice: RocksDB-based vs file-based:

1. Harsha/Satish didn't see significant improvement in performance when they tried RocksDB in their prototype.
2. Other advantages of RocksDB were discussed - snapshots, tooling, checksums etc.
3. As for file-based design, we don't know yet the % of increase in memory, assuming the majority of segments are in remote storage.
4. Currently a single file-based implementation for the whole broker is considered. But this may have some issues, so it can be useful to consider a file per partition.
5. More details needed to be added to the KIP on file management, metadata operations, persisted data format and estimates on memory usage.

Topic ID:

1. KIP-516 PR may land by end of the year, so we should be able to use it in KIP-405.
2. Satish to update KIP with details.

(Notes taken by Kowshik)

10 Nov 2020

- Discussion Recording
- Notes

29 Sep 2020

- Discussion Recording
- Notes
  - Discussed that we can have producerid snapshot for each log segment which will be copied to remote storage. There is already a PR for KAFKA-9393 which addresses similar requirements.
  - Discussed on a case when the local data is not available on brokers, whether it is possible to recover the state from remote storage.
  - We will update the KIP by early next week with
    - Topic deletion design proposed/discussed in the earlier meeting. This includes the schemas of remote log segment metadata events stored in the topic.
    - Producerid snapshot for each segment discussion.
    - ListOffsets API version bump to support offset for the earliest local timestamp.
    - Justifying the rationale behind keeping RLMM and local leader epoch as the source of truth.
    - Rocks DB instances as cache for remote log segment metadata.
    - Any other missing updates planned earlier.

15 Sep 2020

- Discussion Recording
- Notes
  - Discussed the proposed topic deletion lifecycle with and without KIP-516.
    - We will update the KIP with the design details.
    - Jun mentioned that KIP-516 will be available in 3.0 and we can go with the design assuming TopicId support.
  - Discussed on remote log metadata truncation and losing the data of Kafka brokers local storage.
    - We will update KIP on possible approaches and add any possible APIs needed for RemtoeStorageManager(low Priority for now).

01 Sep 2020

- Discussion Recording
- Notes
  - Topic deletion lifecycle
    - Have a separate section
    - Discuss handling deletions when there is no leader.
    - Describe the approaches with and without KIP-516 support.
  - Describe more on how are duplicate log segments in remote storage are handled. This is partly covered in example scenarios but good to describe them in the details section.
  - Discuss more on remote log segment metadata topic truncation.
  - Remote log segment metadata topic event format
    - the event change log approach instead of having an effective event as a message.

        - Behaviour of APIs with remote storage errors.

25 Aug 2020

- Discussion Recording
- Notes
    - KIP is updated with follower fetch protocol and ready to reviewed
    - Satish to capture schema of internal metadata topic in the KIP
    - We will update the KIP with details of different cases
    - Test plan will be captured in a doc and will add to the KIP
    - Add a section "Limitations" to capture the capabilities that will be introduced with this KIP and what will not be covered in this KIP.

Other associated KIPs

KIP-852: Optimize calculation of size for log in remote tier

KIP-917: Additional custom metadata for remote log segment