KIP-712: Shallow Mirroring

Status

Current state: *"Under Discussion"*

Discussion thread: *here*

JIRA: *here*

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

Motivation

In Kafka MirrorMaker v1 (the version we used in house), the mirror process starts with N consumers and one producer. The consumer reads the message from a given topic from the source cluster and sends the data through the producer to write to the destination cluster. There are multiple bytes copying steps along the way:

- A ByteBuffer is allocated from memory pool from NetworkReceive object, the byte buffer corresponds to the fetch request which can contain message batches from different topics from the same source node (https://github.com/apache/kafka/blob/2.3.1/clients/src/main/java/org/apache/kafka/common/memory/MemoryPool.java#L30)
- The byte buffer is iterated / read / de-serialized / copied into consumer's ConsumerRecord object during consumer's pollForFetches step (https://github.com/apache/kafka/blob/2.3.1/clients/src/main/java/org/apache/kafka/common/record/DefaultRecordBatch.java#L333)
- The ConsumerRecord is converted into ProducerRecord in mirror maker and is sent to the producer (https://github.com/apache/kafka/blob/2.3.1/core/src/main/scala/kafka/tools/MirrorMaker.scala#L423)
- The producer will copy/re-serialize the bytes from ProducerRecord into a new byte buffer allocated from producer's batch pool (https://github.com/apache/kafka/blob/2.3.1/clients/src/main/java/org/apache/kafka/common/record/MemoryRecordsBuilder.java#L626)
- The byte buffer from batch pool will be re-grouped together with other topic's byte buffer if they are being sent to the same target node and eventually being written to the socket channel (https://github.com/apache/kafka/blob/2.3.1/clients/src/main/java/org/apache/kafka/common/network/ByteBufferSend.java#L59)

When the message is compressed, the consumer needs to decompress the message and the producer will need to recompress the message, the mirror maker itself does not use the decompressed message. The decompression/recompression step is very costly in the data transfer pipeline and causes extra delay and CPU cost (as high as 40% cpu overhead) and becomes a bottleneck in scaling the mirror maker. The decompression process also uses extensive memory on the fly and we observed up to 2x-10x memory explosion during decompression for some GZIP compressed topic streams. If the partition fetch bytes parameter is not configured properly, the mirror process will trigger OOM when the traffic spikes.

Kafka 0.7 used to have a shallow iterator feature to skip full message decompression, however this feature was disabled in Kafka 0.8 (https://issues.apache.org/jira/browse/KAFKA-732).

The KIP proposes a **shallow mirror** feature which brings back the shallow iterator concept to the mirror process and also proposes to skip the unnecessary message decompression and recompression steps. We argue in many cases users just want a simple replication pipeline to replicate the message as it is from the source cluster to the destination cluster. In many cases the messages in the source cluster are already compressed and properly batched, users just need an identical copy of the message bytes through the mirroring without any transformation or repartitioning.

We have a prototype implementation in house with MirrorMaker v1 and observed **CPU usage dropped from 50% to 15%** for some mirror pipelines.

Proposed change is using MirrorMaker v1 as the reference implementation since that's the product we are using in house. The code change mostly occurs in consumer and producer code paths with some glue code change in mirror maker, it can be adapted to mirror maker v2 by the community as well.

About the name

We name this feature: **shallow mirroring** since it has some resemblance to the old Kafka 0.7 namesake feature but the implementations are not quite the same. '**Shallow**' means 1. we **shallowly** iterate RecordBatches inside MemoryRecords structure instead of deep iterating records inside RecordBatch; 2. We **shallowly** copy (share) pointers inside ByteBuffer instead of deep copying and deserializing bytes into objects. Initially we used another project name: **identity mirroring** to emphasize the fact that the messages mirrored from the source to the destination cluster are identical (same compression, same batch composition). However it seems Cloudera has used the term identity mirror already when introducing Mirror Maker V2 (https://blog.cloudera.com/a-look-inside-kafka-mirrormaker-2/)

Public Interfaces

Briefly list any new interfaces that will be introduced as part of this proposal or any existing interfaces that will be removed or changed. The purpose of this section is to concisely call out the public contract that will come along with this feature.

ConsumerConfig

A new ConsumerConfig boolean option: **fetch.raw.bytes** is added to skip message de-serialization and decompression. When this option is enabled, the consumer will return the underlying ByteBuffer directly as the value field of ConsumerRecord to the caller. The ByteBuffer represents the batch of messages (RecordBatch) fetched from the source broker.

ConsumerRecord

When fetch.raw.bytes is enabled through ConsumerConfig, the value field of ConsumerRecord is the ByteBuffer representing the batch of messages fetched from the source broker. Most of the other fields in ConsumerRecord are null if it doesn't make sense for the whole batch (e.g. key, header fields). The offset field is set to the last message offset of the batch.

For the caller, it will see consumer.poll() return a ConsumerRecord<Void,ByteBuffer>

ProducerConfig

A new ProducerConfig boolean option: **send.raw.bytes** is added to skip message re-serialization and re-compression. When this option is enabled, the caller will pass the ByteBuffer directly through the value portion of ProducerRecord to the producer. The ByteBuffer represents the batch of messages.

Fetch.raw.bytes and send.raw.bytes options need to be turned on together.

ProducerRecord

When send.raw.bytes is enabled through ProducerConfig, the value field of ProducerRecord is the ByteBuffer representing the batch of messages to be forwarded together to the target broker. Most of the other fields in ProducerRecord are set to null if it doesn't make sense for the whole batch (e.g. key, header fields).

For the caller, it will call producer.send with ProducerRecord<Void,ByteBuffer>

MirrorMaker (v1) command line option

A new command line option: **use.raw.bytes** is added. When this option is enabled, the mirror maker will set the fetch.raw.bytes option for the consumer and send.raw.bytes option on the producer

MirrorMakerMessageHandler (v1)

The original MirrorMaker's message handler plugin: MirrorMakerMessageHandler is hard coded to handle the message of type [Array[Byte],Array[Byte]]. We added a new MirrorMakerRawBytesMessageHandler to handle the message of type [Void,ByteBuffer]

ProduceRequest

Up to ProduceRequest V2, a ProduceRequest can contain multiple batches of messages stored in the record_set field, but this was disabled in V3. We are proposing to bring the multiple batches feature back to improve the network throughput of the mirror maker producer when the original batch size from source broker is too small.

The following ProduceRequest/Response V8 are essentially the same as the ProduceRequest/Response V7 (current version) schema wise, but the record_set field can contain multiple batches.

```
ProduceRequest (Version: 8) => transactional_id acks timeout topic_data
    transactional_id => STRING
    acks => INT16
    timeout => INT32
    topic_data => [topic data]
        topic => STRING
        data => [partition_id record_set]
            partition_id => INT32
            record_set => DocumentType of MemoryRecords

ProduceResponse (Version: 8) => responses throttle_time_ms
    responses => [topic partition_responses]
        topic => STRING
        responses => [partition_id error_code base_offset log_append_time]
            partition_id => INT32
            error_code = INT16
            base_offset => INT64
            log_append_time => INT64
    throttle_time_ms => INT32
```

Proposed Changes

In our mirror pipelines, the source and destination cluster have the same set of topic/partitions and same configurations. Most of the messages on the source cluster are already compressed. We want the mirror maker to work as an efficient pass-through pipe to carry the bytes as it is from the source cluster to the destination cluster without any re-processing and we want the ProduceRequest message going to the destination broker almost the same as the FetchResponse message read from the source broker.

Raw Bytes Mode

In order to make this a streamlined pipe, we want the mirror to transfer raw bytes rather than the de-serialized/de-compressed consumer record. In a FetchResponse, each topic-partition contains a MemoryRecords ByteBuffer which is composed of a series of RecordBatch. Each RecordBatch would contain a batch of records. Decompression would be triggered when RecordBatch.streamingIterator() is called. So instead of iterating through and decompressing (and later de-serializing) each record, we would just return the ByteBuffer representing RecordBatch itself as a 'record' back to the fetcher and later back to the consumer. So when Consumer.poll() is called, it will get back one ConsumerRecord<Void,ByteBuffer> where the ByteBuffer represents the bytes corresponding to RecordBatch. Note that the record key is essentially null in this case since we are dealing with the batch as a unit now.

Later the mirror maker would pass this ByteBuffer into ProducerRecord<Void,ByteBuffer> and call Producer.send. Inside producer code path, Producer.send() RecordAccumulator.append() ProducerBatch.tryAppend() MemoryRecordsBuilder.append(). In MemoryRecordsBuildler.append, we will append the bytes from passed-in ByteBuffer directly into the buffer allocated from RecordAccumulator's BufferPool. The remaining code path remains intact: MemoryRecordsBuilder.build() will be called to return a MemoryRecords which contains the byte buffer and be passed on to create the producer's ProduceRequest message.

Changes in consumer config and ConsumerRecord

A new ConsumerConfig option: **fetch.raw.bytes** is added to skip message de-serialization and decompression.

When fetch.raw.bytes is enabled through ConsumerConfig, the value field of ConsumerRecord is the ByteBuffer representing the batch of messages fetched from the source broker. Most of the other fields in ConsumerRecord are null if it doesn't make sense for the whole batch (e.g. key, header fields). The offset field is set to the last message offset of the batch.

For the caller, it will see consumer.poll() returns a ConsumerRecord<Void,ByteBuffer>

Changes in consumer code path

When the client makes a call: Consumer.poll(), the following code path will be invoked: KafkaConsumer.poll() KafkaConsumer.pollForFetches() Fetcher.fetchRecords() Fetcher.nextFetchedRecord() RecordBatch.streamingIterator()

Rather than explode RecordBatch and decompress/iterate through all records, we would just return a RawBytesBatchRecord (a new class implementing Record interface, containing the byte buffer corresponding to this batch).

This special Record class would trigger the return code path to skip deserialization and return a ConsumerRecord<Void,ByteBuffer>

Dealing with the case when consumer fetches from the middle of the batch

Each RecordBatch inside FetchResponse's record_set field 1:1 corresponds to the original batches when they were stored inside the source broker's disk system. When the consumer first fetches in raw.bytes mode, the first offset it was fetching might be in the middle of a batch. In this case, we would need to filter out the records whose offset are smaller than the consumer's starting offset. We would create a new byte buffer which only contains records whose offset are in the fetch range and return this smaller byte buffer instead. This would only happen for the first fetch from mirror maker (and mirror maker would commit offsets based on batch's boundary afterwards), so the performance impact is minimal.

Changes in producer config and ProducerRecord

A new ProducerConfig option: **send.raw.bytes** is added to skip message re-serialization and re-compression.

When send.raw.bytes is enabled through ProducerConfig, the value field of ProducerRecord is the ByteBuffer representing the batch of messages to be forwarded together to the target broker. For the caller, it will call producer.send with ProducerRecord<Void,ByteBuffer>

Changes in producer code path

The client will call Producer.send(ProducerRecord<Void,ByteBuffer>) which invokes the following code path: KafkaProducer.send() KafkaProducer. doSend() RecordAccumulator.append() ProducerBatch.tryAppend() MemoryRecordsBuilder.append()

The existing code at this point will append each new incoming record to the builder and later MemoryRecordsBuilder.build() will be called to calculate and set the batch header fields and generate the MemoryRecords byte buffer. In the new code, when the incoming data is already a ByteBuffer, we would skip all the append and build phases to directly copy the bytes from the incoming ByteBuffer into the buffer in the builder.

Changes in mirror maker v1

A new command line option: **use.raw.bytes** is added. When this option is enabled, the mirror maker will set the fetch.raw.bytes option for the consumer and send.raw.bytes option on the producer.

When use.raw.bytes mode is on, the mirror is reading/transferring a batch a time and it will also commit offset based on batch boundary. The ConsumerRecord's offset field in the batch case represents the offset of the last record in that batch.

The original MirrorMaker's message handler plugin: MirrorMakerMessageHandler is hard coded to handle the message of type [Array[Byte],Array [Byte]]. We added a new MirrorMakerRawBytesMessageHandler to handle the message of type [Void,ByteBuffer]

Multiple Batches in ProduceRequest

In the first iteration of this project, we just set one RecordBatch into RecordAccumulator's buffer and the resulting ProduceRequest message only contains one batch. We noticed the resulting networking performance was not great especially when the original batch is small. The FetchResponse message from source broker node often have byte buffer in the size of MB since it can contains multiple batches but the outgoing ProduceRequest message going into destination broker often has size of KB, the smaller outbound message caused many more round trips between mirror maker node and destination broker node and sequential processing nature of those ProduceRequest messages on broker side drags down the performance of the mirror pipe.

Actually there is no reason why the ProduceRequest message can only contain one batch. ProduceRequest message up to V2 can contain multiple batches. In ProduceRequest message format V3, the multiple batching is disabled but this creates the disparity between ProduceRequest message and FetchResponse message, the latter still supports multiple batches. And if we look at the key data structure in producer and consumer code path: MemoryRecords, a MemoryRecords object contains a series of batches.

We decide to give multiple batches a try and it doesn't seem too hard to support that mode. On the client side, in the producer's code path: MemoryRecordsBuilder.append() we allow continuously appending the byte buffer from subsequent RecordBatch, the resulting MemoryRecords contains multiple batches. On the broker side, we removed the check which disallows multiple batches in ProduceRequest.

Handling multiple batches and skip message conversion in the broker

Most of the code changes in this KIP occurs on client side, but we also make a few small changes on broker side to support multiple batches for ProduceRequest and skip message conversions:

- In ProduceRequest.validateRecords(), remove the check for multiple batches for V3 message
- LogValidator.validateMessagesAndAssignOffsetsCompressed(), we would loop through the batches from MemoryRecords (instead of assuming only one batch)

Compatibility, Deprecation, and Migration Plan

- The new mirror behavior is only invoked when fetch.raw.bytes and send.raw.bytes options are turned on. The existing user code should be able to work as it is.
- If the user has their own implementation of MirrorMaker v1's MirrorMakerMessgeHandler, they would need to provide implementation for the new MirrorMakerRawBytesMessageHandler; If the user is using the default MirrorMaker MessageHandler, no code change is needed
- ProduceRequest message now allows for multiple batches in record_set field

Rejected Alternatives

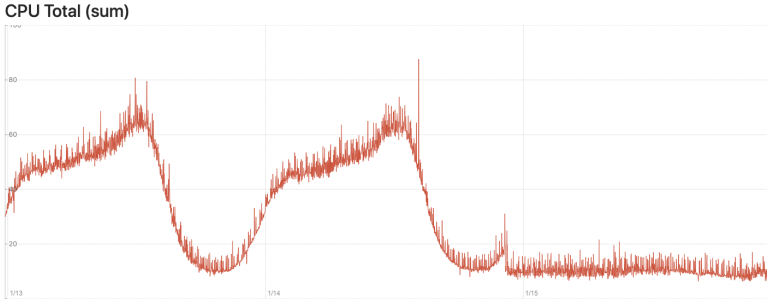
- One alternative is the current mirror implementation where the message is de-serialized, de-compressed and then re-serialized and re-compressed. Although it is inefficient on the surface, the current implementation has its own merit. It helps in the case when the original messages on the source broker is not compressed or has a very small number of messages in a batch, the mirroring brings in the second chance of compressing the message or creating a bigger batch of messages. And the current implementation also works better if the user has their own message handler to further process the message since they are dealing with a full deserialized ConsumerRecord object (instead of raw ByteBuffer). The current implementation would be more appropriate in those cases and the user can choose to use the old mirror behavior by not setting use.raw.bytes config (which is off by default).
- Another alternative is just to skip the mirror maker, write a new low-level component which reads the FetchResponse message directly from the source broker, break the message per topic-partition and re-group the messages into ProduceRequest per target broker's topic-partition grouping. This is going to be more efficient and will have less incision into the existing code path. However we would have to add code to deal with workload balancing and rebalancing which is currently handled by the high level consumer group.

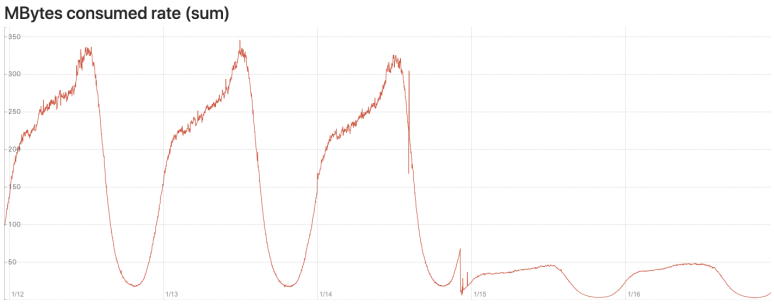
Future Work

The prototype was implemented using and tested with MirrorMaker v1 since we are still on Kafka v2.3 where MirrorMaker v2 is not included and we are using MirrorMaker v1 for our production mirroring pipelines. Porting this solution to MirrorMaker v2 should be straightforward since most of the code changes are in the code path of Consumer, Producer and Broker.

Performance Chart

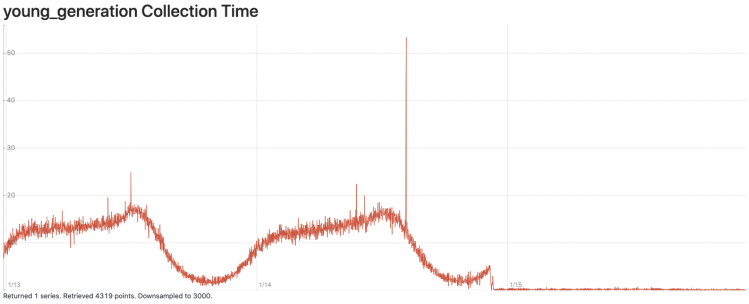
Here are some performance charts to show the CPU and memory usage before and after we deploy the shallow mirroring feature. The switch to shallow mirroring occurred shortly before 1/15.

- CPU difference before and after the feature implementation
- 

- Memory Consumption Related to incoming messages before and after
- 

- GC collection time (young generation) before and after

young_generation Collection Time



- Returned 1 series. Retrieved 4319 points. Downsampled to 3000.