KIP-283: Efficient Memory Usage for Down-Conversion

Status

Current state: Accepted

Discussion thread: *here*

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

Released: 2.0.0

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

Motivation

Kafka typically uses zero-copy optimization for transferring data from the file system cache to the network socket when sending messages from brokers to consumers. This optimization works only when the consumer is able to understand the on-disk message format.

An older consumer would expect an older message format than what is stored in the log. Broker needs to convert the messages to the appropriate older format that the consumer is able to understand (i.e. down-convert the messages). In such cases, the zero-copy optimization does not work as we need to read the messages from the file system cache into the JVM heap, transform the messages into the appropriate format, and then send them over the network socket. We could end up in situations where this down-conversion process causes Out Of Memory because we end up copying lot of data into the JVM heap.

Following are the goals of this KIP:

1. Reduce overall footprint of the down-conversion process, both in terms of memory usage and the time for which the required memory is held.
2. As we cap the memory usage, we'd inadvertently affect the throughput and latency characteristics. Lesser the amount of memory available for down-conversion, fewer messages can batched together in a single reply to the consumer. As fewer messages can be batched together, the consumer would require more round-trips to consume the same number of messages. We need a design that minimizes any impact on throughput and latency characteristics.

Public Interfaces

Broker Configuration

A new broker configuration will be provided to completely disable down-conversion on the broker to satisfy `FetchRequest` from client. This configuration provides an added measure to the optimizations discussed in this KIP.

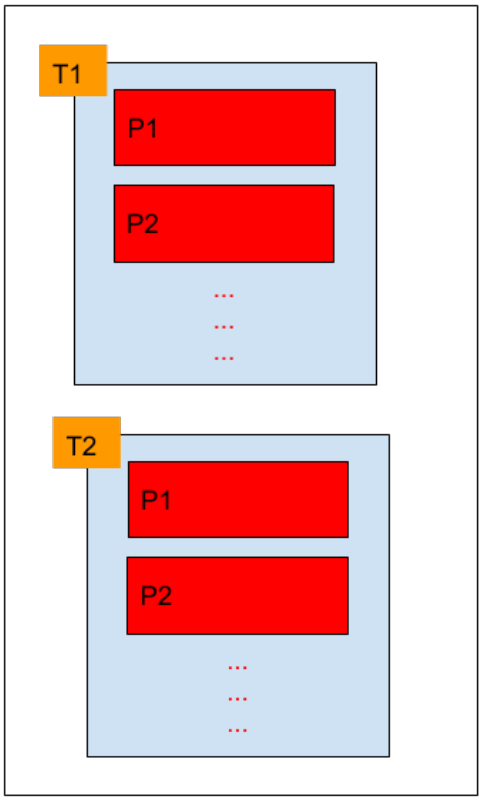
- Topic-level configuration: `message.downconversion.enable`
- Broker-level configuration: `log.message.downconversion.enable`
- Type: Boolean
- Possible Values: True / False
- Explanation: Controls whether down-conversion of messages is enabled to satisfy client `FetchRequest`. If true, broker will down-convert messages when required. If set to false, broker will disable down-conversion of messages and will send `UnsupportedVersionException` in response to any client `FetchRequest` that requires down-conversion.
- Default Value: True

Proposed Changes

Message Chunking

We will introduce a message chunking approach to reduce overall memory consumption during down-conversion. The idea is to lazily and incrementally down-convert message "chunks" in a batched fashion, and send the result out to the consumer as soon as the chunk is ready. The broker continuously down-converts chunks of messages and sends them out, until it has exhausted all messages contained in the `FetchResponse`, or has reached some predetermined threshold. This way, we use a small, deterministic amount of memory per `FetchResponse`.

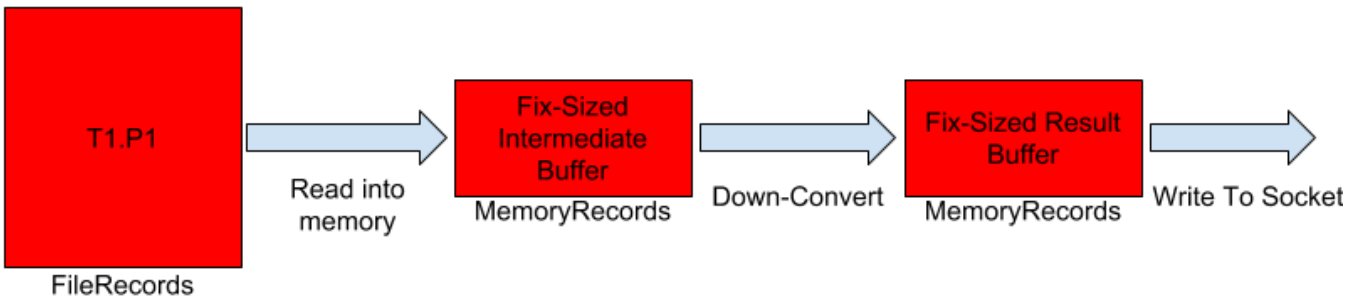
The diagram below shows what a `FetchResponse` looks like (excluding some metadata that is not relevant for this discussion). The red boxes correspond to the messages that need to be sent out for each partition. When down-conversion is not required, we simply hold an instance of `FileRecord`s for each of these which contains pointers to the section of file we want to send out. When sending out the `FetchResponse`, we use zero-copy to transfer the actual data from the file to the underlying socket which means we never actually have to copy the data in userspace.



When down-conversion is required, each of the red boxes correspond to a `MemoryRecords`. Messages for each partition are read into JVM heap, converted to the appropriate format, and we hold on to this memory containing converted messages till the entire `FetchResponse` is created and subsequently sent out. There are couple of inefficiencies with this approach:

1. The amount of memory we need to allocate is proportional to all the partition data in `FetchResponse`.
2. The memory is kept referenced for an unpredictable amount of time - `FetchResponse` is created in the I/O thread, queued up in the `responseQueue` till the network thread gets to it and is able to send the response out.

The idea with the chunking approach is to tackle both of these inefficiencies. We want to make the allocation predictable, both in terms of amount of memory allocated as well as the amount of time for which the memory is kept referenced. The diagram below shows what down-conversion would look like with the chunking approach. The consumer still sees a single `FetchResponse` which is actually being sent out in a "streaming" fashion from the broker.



The down-conversion process works as follows:

1. Read a set of message batches into memory.
2. Down-convert all message batches that were read.
3. Write the result buffer to the underlying socket.
4. Repeat till we have sent out all the messages, or have reached a pre-determined threshold.

With this approach, we need temporary memory for to hold a batch of down-converted messages till they are completely sent out. We will limit the amount of memory required by limiting how many message batches are down-converted at a given point in time. A subtle difference also is the fact that we have managed to delay the memory allocation and the actual process of down-conversion till the network thread is actually ready to send out the results. Specifically, we can perform down-conversion when `Records.writeTo` is called.

Although we have increased some amount of computation and I/O for reading log segments in the network thread, this is not expected to be very significant to cause any performance degradation. We will continue performing non-blocking socket writes in the network thread, to avoid having to stall the network thread waiting for socket I/O to complete.

Messages that require lazy down-conversion are encapsulated in a new class called `LazyDownConvertedRecords`. `LazyDownConvertedRecords#writeTo` will provide implementation for down-converting messages in chunks, and writing them to the underlying socket.

At a high level, `LazyDownConvertRecords` looks like the following:

```
public class LazyDownConvertRecords implements Records {
    /**
     * Reference to records that will be "lazily" down-converted. Typically FileRecords.
     */
    private final Records records;

    /**
     * Magic value to which above records need to be converted to.
     */
    private final byte toMagic;

    /**
     * The starting offset of records.
     */
    private final long firstOffset;

    /**
     * A "batch" of converted records.
     */
    private ConvertedRecords<? extends Records> convertedRecords = null;

    ...
}
```

Determining Total Response Size

Kafka response protocol requires the size of the entire response in the header. Generically, `FetchResponse` looks like the following:

```
FetchResponse => Size CorrelationId ResponseDataAndMetadata
  Size => Int32
  CorrelationId => Int32
  ResponseDataAndMetadata => Struct

where
Size = HeaderSize + MetadataSize + (TopicPartitionDataSize)
```

`HeaderSize` and `MetadataSize` is always fixed in size, and can be pre-computed. `TopicPartitionDataSize` is a variable portion which depends on the size of the actual data being sent out. Because we delay down-conversion till `Records.writeTo`, we do not know the exact size of data that needs to be sent out.

To resolve this, we will use an estimate S_{pre} (defined below) as opposed to the actual post-downconversion size (S_{post}) for each topic-partition. We also commit to send out exactly S_{pre} bytes for that particular partition. In terms of the above equation, the size will be computed as:

```
Size = HeaderSize + MetadataSize + (Spre)

where Spre = max(size_pre_downconversion, size_first_batch_after_downconversion)
```

Given this, we have three possible scenarios:

1. **S_{pre} = Spost:** This is the ideal scenario where size of down-converted messages is exactly equal to the size before down-conversion. This requires no special handling.
2. **S_{pre} < S_{post}:** Because we committed to and cannot write more than S_{pre}, we will not be able to send out all the messages to the consumer. We will down-convert and send all message batches that fit within S_{pre}.
3. **S_{pre} > S_{post}:** Because we need to write exactly S_{pre}, we append a "fake" message at the end with maximum message size (= Integer. MAX_VALUE). Consumer will treat this as a partial message batch and should thus ignore it.

Ensuring Consumer Progress

To ensure that consumer keeps making progress, Kafka makes sure that every response consists of at least one message batch for the first partition, even if it exceeds `fetch.max.bytes` and `max.partition.fetch.bytes` (KIP-74). When S_{pre} < S_{post} (second case above), we cannot send out all the messages and need to trim message batch(es) towards the end. Because S_{pre} is at least as big as the size of first batch after down-conversion, we are guaranteed that we send out at least that one batch.

Determining number of batches to down-convert

A limit will be placed on the size of messages down-converted at a given point in time (what forms a "chunk"). A chunk is formed by reading a maximum of 16kB of messages. We only add full message batches to the chunk. Note that we might have to exceed the 16kB limit if the first batch we are trying to read is larger than that.

Pros

- Fixed and deterministic memory usage for each down-conversion response.
- Significantly reduced duration for which down-converted messages are kept referenced in JVM heap.
- No change required to existing clients.

Cons

- More computation being done in network threads.
- Additional file I/O being done in network threads.
- Complexity - expect this to be a big patch, but should be able to keep it abstracted into its own module so that it does not cause too much churn in existing code.
- Could be viewed as being somewhat "hacky" because of the need the need to pad "fake" messages at the end of the response.

Ability to Block Older Clients

Even though the chunking approach tries to minimize the impact on memory consumption when down-conversions need to be performed, the reality is that we cannot completely eliminate its impact on CPU and memory consumption. Some users might want an ability to completely block older clients, such that the broker is not burdened with having to perform down-conversions. We will add a broker-side configuration parameter to help specify the minimum compatible consumer that can fetch messages from a particular topic (see the "Public Interfaces" section for details).

Compatibility, Deprecation, and Migration Plan

There should be no visible impact after enabling the message chunking behavior described above. One thing that clients need to be careful about is the case where the total response size is greater than the size of the actual payload (described in **S_{pre} > S_{post}** scenario of the chunking approach); client must ignore any message who size exceeds the size of the total response.

Testing Strategy

There are two main things we want to validate with the changes brought in by this KIP.

1. Efficiency of the chunking approach
    a. Did we improve overall memory usage during down-conversion?
    b. Estimate of how much memory is consumed for each down-conversion the broker handles.
    c. How many concurrent down-conversions can the broker support?
2. Effect on consumer throughput for various chunk size

The following describes the test setup for each of the points above and the corresponding findings.

Efficiency of Chunking Approach

The aim of the test was to prove that we have a finite bound on the amount of memory we consumer during down-conversion, regardless of the fetch size. The test setup is described in detail below.

Test Setup:
- Java heap size = 200MB
- 1M messages, 1kB each ==> 1GB of total messages
- Split into 250 partitions ==> approximately 3.5MB per partition
- Single consumer with `fetch.max.bytes` = 250MB and `max.partition.fetch.bytes` = 1MB
- Each fetch consumes min(1MB*250, 250MB) = 250MB

Success criteria:
- Must always run out of memory if not using lazy down-conversion
- Must never run out of memory if using lazy down-conversion

Findings:
- Without the chunking approach, we down-convert all messages for all partitions, resulting in us running out of heap space.
- With the chunking approach, down-conversion consumes maximum of the chunked size at a time (which was set to 128kB). This keeps the memory usage both deterministic and finite, regardless of the fetch size.

Effect on consumer throughput

The aim of this test was to study the effect on throughput as we vary the chunk size, and to find the optimal chunking size.

Test Setup:
- 1 topic
- 12 partitions
- 10M messages, 1kB each
- 1 consumer
- Consume messages from start to end 10 times

The following table outlines the findings:

Chunk Size	Average Throughput (MBPS)
16kB	136.95
32kB	167.04
64kB	181.92
128kB	197.72
256kB	181.25
512kB	180.41
1MB	176.64

The average throughput without the chunking approach (i.e. without this KIP) was found to be 178.9MBPS. Given this, the default chunk size will be configured to 128kB.

Rejected Alternatives

We considered several alternative design strategies but they were deemed to be either not being able to address the root cause by themselves, or being too complex to implement.

Alternate Chunked Approach

The "hacky"ness of the chunked approach discussed previously can be eliminated if we allowed the broker to break down a single logical `FetchResponse` into multiple `FetchResponse` messages. This means if we end up under-estimating the size of down-converted messages, we could send out two (or more) `FetchResponse` to the consumer. But we would require additional metadata so that the consumer knows that the multiple `FetchResponse` actually constitute a single logically continuous response. Because tagging the additional metadata requires changes to `FetchResponse` protocol, this will however not work for existing consumers.

We could consider this option as the eventual end-to-end solution for the down-conversion issue for future clients that require down-conversion.

Native Heap Memory

Kafka brokers are typically deployed with a much smaller JVM heap size compared to the amount of memory available, as large portions of memory need to be dedicated for the file system cache.

The primary problem with down-conversion today is that we need to process and copy large amounts of data into the JVM heap. One way of working around having to allocate large amount of memory in the JVM heap is by writing the down-converted messages into a native heap.

We can incrementally read messages from the log, down-convert them and write them out to native memory. When the entire `FetchResponse` has been down-converted, we can then put the response onto the network queue like we do today.

Pros

- Easy to implement.

Cons

- Need to devise effective way to reuse allocated native heap space (memory pooling).
- Tricky to design for cases where required memory is not immediately available.
    - Could lead to consumer starvation if we wait too long for memory to become available, or if we never get the required amount of memory.
- Same amount of memory still consumed in aggregate, just not from the JVM heap. Memory will also not be freed till GC calls finalize.
- We need some estimate of how much memory to allocate before actually performing the down-conversion.

Configuration for Maximum Down-Conversion Memory Usage

Expose a broker-side configuration *message.max.bytes.downconvert*, which is the maximum amount of memory the broker can allocate to hold down-converted messages. This means that a message fetch that requires down-conversion would be able to fetch maximum of *message.max.bytes.downconvert* in a single fetch. If required memory is not available because it is already being consumed by other down-converted messages, we block the down-conversion process till sufficient memory becomes available.

The implementation for this could be based on the *MemoryPool* model, which essentially provides an interface to manage a fixed amount of memory. Each message that requires down-conversion requests for memory from the underlying *MemoryPool* and releases back to the pool once the message is sent out to the consumer.

Pros

- Likely straight-forward in terms of implementation.
- We have an upper-bound on how much memory can be consumed by down-conversion on the broker.

Cons

- Determining what *message.max.bytes.downconvert* should be set to could be a challenge, as setting it too low could affect latency, while setting it too high could cause memory pressure for other modules that share the JVM heap.
- Another knob for administrators to worry about!
- Need to devise effective way to reuse allocated native heap space (memory pooling).
- Tricky to design for cases where required memory is not immediately available.

Compression

To reduce the overall amount of memory consumed by down-converted messages, we could consider compressing them (if they are not already compressed to begin with).

Pros

- Reduced overall memory usage for down-converted messages.

Cons

- Choosing the correct compression format would require some experimentation.
- This by itself would not resolve the root cause because the compressed batch of messages might still end up causing OOM.