Discussion: Memory Management in Kafka Streams

There have been some questions and discussions about how to efficiently let users to configure their memory usage in Kafka Streams since 0.10.0 release, and how that will affect our current development plans regarding caching, buffering, and state store management, etc. In this page we summarize the memory usage background in Kafka Streams as of 0.10.0, and discuss what would be the "end goal" for Kafka Stream's memory management. This is not used as an implementation design and development plan for memory management, but rather as a guidance for related feature developments that may be correlating to the memory usage.

Background

There are a few modules inside Kafka Streams that allocate memory during the runtime:

1. Kafka Producer: each thread of a Kafka Streams instance maintains a producer client. The client itself maintains a buffer for batching records that are going to be sent to Kafka. This is completely controllable by producer's `buffer.memory` config.

2. Kafka Consumer: each thread of a Kafka Streams instance maintains two consumer clients, one for normal data fetching and another one for state store replication and restoration only. Each client buffers fetched messages before they are returned to user from the **poll** call. Today this buffer is not controllable yet, but in the near future we are going to add similar memory bound controls like we have in producers: KAFKA-2045.

3. Both producer and consumer also have separate TCP send / receive buffers that are not counted as the buffering memory, which are controlled by the send.buffer.bytes / receive.buffer.bytes configs; these are usually small (100K) and hence neglected most of the time.

4. Triggering based Caches: as summarized in KIP-63, we will be adding a cache for each of the aggregation and KTable.to operators, and we are adding a StreamsConfig to bound the total number of bytes used for all caches. BUT we are caching them as deserialized objects in order to avoid serialization costs.

5. Deserialized Objects Buffering: within each thread's running loop, after the records are returned in raw bytes from **consumer.poll**, the thread will deserialize each one of them into typed objects and buffer them, and process them one record at-a-time. This is mainly used for extracting the timestamps (which may be in the message's value payload) and for reasoning about the streams-time to determine which stream to process next (i.e. synchronizing streams based on their current timestamps, see this for details).

6. Persistent State Store Buffering: This is related to KIP-63. Currently we are using RocksDB by default as persistent state stores for stateful operations such as aggregation / joins, and RocksDB have their own buffering and caching mechanism which allocate memory both off-heap and on-heap. And RocksDB has its own configs that controls their sizes (we plan to expose these configs separately from StreamsConfig: KAFKA-3740), to name a few:
   - block_cache_size: amount of cache in bytes that will be used by RocksDB. NOTE this is off-heap.
   - write_buffer_size: the size of a single memtable in RocksDB.
   - max_write_buffer_number: the maximum number of memtables, both active and immutable.

     So a rough calculation about the amount of memory: block_cache_size + write_buffer_size * max_write_buffer_number.

Memory Management: The End Goal

In the ideal world, Kafka Streams should provide very simple configuration for its memory management. More concretely, for example, users should be able to just specifying a single config value that bounds the total usage of 1) + 2) + 3) + 4) + 5) above, for example:

```
streams.memory.bytes (denoted as Total)
```

while keeping in mind that it needs to be at least be larger than

```
producer.memory.bytes + consumer.memory.bytes
```

which represent case 1) and 2) above, and can also be specified by the user through the StreamsConfig, but in practice they may just be using the default values.

And hence if users start their Streams application in a container with bounded memory usage as **X**, they know that their coded application can use up to the amount of memory allowed by the container minus total allocable Streams library usage, i.e. **X - S**. And even under task migration scenarios upon failures, or rebalancing, the immigrated tasks which will then allocate memory for its own caching and state stores, etc, will not suddenly increase the libraries memory usage since its total is still bounded, and hence not causing OOM (note that in case of task migration due to one instance failure, without memory bounding it may cause cascading OOMs, which is really bad user experience).

With this end goal in mind, now let's see how we should bound the memory usage for the above cases, especially 3), 4) and 5).

Triggering based Caches

The total memory used for this part (denoted as **Cache**) can be calculated as:

```
SUM_{all threads within an KafkaStreams instance} (SUM_{all tasks allocated to the thread} (SUM_{all caches created within this task's topology} (#.bytes in this cache)))
```

NOTE that:

- This total is dynamic from rebalance to rebalance since the tasks assigned to each thread can change, and hence the corresponding sub-topology's number of caches can change too.
- Because we have triggering-based caches on top of all RocksDB instances for aggregations, we are effectively caching the records twice (one cache on top of RocksDB, and one cache inside RocksDB as Memtables). We have this extra caching in objects originally only for reducing serialization costs.

Future Plan Proposal

This should be considered as part of KIP-63.

We know that for deserialized objects, their size in bytes are hard to estimate accurately without serializing them to bytes. Hence we should consider just caching these values in terms of byte arrays and always pay the serialization / deserialization costs for better memory management.

And in the future we will allow users to configure which state stores they are going to use for their stateful operations: it can be persistent or in-memory, and then:

- We only need in-memory caching if persistent stores are used for aggregates, which will introduce extra serde costs as mentioned above.
- If the state stores used are already in-memory (and this should be in deserialized objects), we do not need the caching in bytes any more, while we still keep the dirty map for triggered flushing.

Deserialized Objects Buffering

The total memory used for this part (denoted as **Buffer**) can be calculated as:

```
SUM_{all threads within an KafkaStreams instance} (SUM_{all tasks allocated to the thread} (SUM_{partitions assigned to the task} (#.bytes buffered for that partition)))
```

Today we have a config

`buffered.records.per.partition`

that controls how many records we would buffer before pausing the fetching on that partition, but that 1) does not restrictedly enforce the upper limit on the number of records, and 2) number of deserialized records does not imply #. bytes.

Future Plan Proposal

Assuming that in the future most users will define record timestamps to be the timestamp on the message metadata field, and for the rare case where user's specify a different timestamp extractor we are willing two pay the deserialization cost twice just for getting the source timestamp, then we can keep this buffer in raw bytes as well: i.e. if the default record timestamp extractor is used, we just get the raw bytes records from consumer.poll and extract their timestamps; if other timestamp extractor is used, we deserialize the record to get the timestamp, and throw away the deserialized records but still keep the raw bytes in its buffer. In this case, we can change the config to:

`buffered.bytes.per.partition`

Persistent State Store Buffering

The total memory used for this part (denoted as **Store**) as:

```
SUM_{all threads within an KafkaStreams instance} (SUM_{all tasks allocated to the thread} (SUM_{all RocksDB stores in the task} (total #.bytes allocated for this RocksDB)))
```

Future Plan Proposal

For advanced users who have good understandings about RocksDB configs, they should still be able to specify these config values such as 1) block cache size, 2) Memtable buffer size, 3) number of Memtables, 4) etc for a single KafkaStreams instance; and if no user-specified values are provided some default values will be provided. BUT for some of these configs like block cache size, it should be a per Kafka Streams instance config instead of a per RocksDB config, and hence the Streams library should divide its values among the threads / tasks / RocksDB instances dynamically.

And also as a side note, if we are using bytes in our own caching layer as proposed above, then we should try to reduce the usage of RocksDB's own Memtable by default as it effectively have less benefits additionally.

Summary

So putting it all together, here is the proposal of Kafka Streams to reason about its memory usage:

- The user specified total amount of memory **Total** of a Kafka Streams instance is always divided evenly to its threads upon starting up the instance, whose number is static throughout its life time.
- Within a single stream thread, the total memory **Total / numThreads** will first be subtracted by the reserved memory for its producer (denoted as **Producer**) and consumer (denoted as **Consumer**) client usage, whose values are also static throughout the thread's life time.
- For the rest of usable memory **Total / numThreads - Producer - Consumer**, it is dynamically allocated upon each rebalance:
  - Every time upon a rebalance, when the assigned tasks are created, the thread will first extract the memory by the amount of buffering needs (**Buffer**), calculated as *buffered.bytes.per.partition * total.number.partitions*.
  - Then it will extract the amount of memory used for all its persistent state stores (**State**), calculated by different store's specific equations, for example for RocksDB it is calculated as *block_cache_size + write_buffer_size * max_write_buffer_number*.
  - If the rest amount of memory **Total / numThreads - Producer - Consumer - Buffer - State** is already negative, then we should log WARNING that there may not be not enough memory for this instance.
  - Otherwise, the rest amount of memory is allocated for caching needs (**Cache**), and multiple caches will try to dynamically allocate memory from this buffer pool, and possibly flushing if it is about to be exhausted, as we mentioned above.

NOTE that the caveat of this approach is that the amount of **Buffer** and **State** can increase with the number of tasks assigned to this instance's threads, and hence we may not actively guarding against the cascading OOMs as we mentioned above. As Jay Kreps suggested in an off-line discussion, one way to think of this issue is the following:

- Among all the memory usage listed above, **Producer** and **Consumer** are "*per-thread*", and since #.threads are static throughout the Kafka Streams life time, their total memory usage is also static and hence is easily bounded.
- **State**, **Buffer** and **Cache** are *"per-task"*, which can change dynamically from rebalance to rebalance as #.tasks assigned to the threads of the Kafka Streams instance can change over time.
  - Therefore calculating their usage in a *"bottom-up"* manner by letting users specify its upper bound per-task / per-store / per-partition will not be able to bound the total memory they use, and hence upon task migration cascading OOMs could happen.
  - Instead, we should bound their usage in a "top-down" manner, i.e. we should calculate the per-task / per-store / per-partition configs based on the total allocable memory (i.e. **Total / numThreads - Producer - Consumer**) and the #.tasks / etc upon rebalancing dynamically, for example *buffered.bytes.per.partition* and RocksDB's *block_cache_size.*
- Regarding **Buffer** specifically, currently it is based on a per-partition config (buffered.records.per.partition), but since its usage is only for reasoning about the stream time, and its deserializing raw bytes are bounded by Consumer already, we should consider configuring it also at the global level, for example replacing buffered.records.per.partition with buffered.records.bytes.

Open Questions

1. Should we buffer records or bytes for **Buffer**? The pros of buffering records is avoid deserialization overheads, but the cons are expensive record size estimates.
2. Should we buffer records or bytes for **Cache**? Similar trade-offs as above, but in addition that given the scenarios if there is already a state store underneath with its own block caching in bytes, should we consider removing this cache at all and only relying on the state store (RocksDB)'s own block caching, while only keeping the dirty map in bytes and pay the get() values for all dirty keys and deserialization upon flushing?

The above questions can probably be better answered by inducting some benchmark experiments comparing, for example https://github.com/jbellis/jamm f or estimating record sizes with serialization / deserialization costs.