KIP-41: KafkaConsumer Max Records

Status

Current state: *Adopted*

Discussion thread: http://mail-archives.apache.org/mod_mbox/kafka-users/201512.mbox/%3CCAL%2BArfWNfkpymkNDuf6UJ06CJJ63XC1bPHeT4TSYXKjSsOpu-Q%40mail.gmail.com%3E (user mailing list)

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

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

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

Motivation

The KafkaConsumer API centers around the poll() API which is intended to be called in a loop. On every iteration of the loop, poll() returns a batch of records which are then processed inline. For example, a typical consumption loop might look like this:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(1000);
  records.forEach(record -> process(record));
  consumer.commitSync();
}
```

In addition to fetching records, poll() is responsible for sending heartbeats to the coordinator and rebalancing when new members join the group and old members depart. The coordinator maintains a timer for every member in the group which is reset when a heartbeat is from that member is received. If no heartbeat is received before the expiration of a configurable session timeout, then the member is kicked out of the group and its partitions are reassigned to other members. If this happens while the consumer is processing a batch of records, there is no direct way for the consumer to stop processing and rejoin the group. Instead, the loop will finish processing the full batch and only find out afterwards that it has been kicked out of the group. This can lead to duplicate consumption since the partitions will already have been reassigned to another member in the group before offsets can be committed. In the worst case, if the processing time for record batches frequently approaches the session timeout, this can cause repeated rebalances and prevent the group from making progress.

Users can mitigate this problem by 1) increasing the session timeout, and 2) reducing the amount of data to process on each iteration of the loop. Increasing the session timeout is what we've recommended thus far, but it forces the user to accept the tradeoff of slower detection time for consumer failures. It's also worth noting that the time to rebalance depends crucially on the rate of heartbeats which is limited by the processing time in each iteration of the poll loop. If it takes one minute on average to process records, then it will also generally take one minute to complete rebalances. In some cases, when the processing time cannot be easily predicted, this option is not even viable without also adjusting the amount of data returned. For that, we give users a "max.partition.fetch.bytes" setting which limits the amount of data returned for each partition in every fetch. This is difficult to leverage in practice to limit processing time unless the user knows the maximum number of partitions they will consume from and can predict processing time according to the size of the data. In some cases, processing time does not even correlate directly with the size of the records returned, but instead with the count returned. It is also difficult to deal with the effect of increased load which can simultaneously increase the amount of data fetched and increase the amount of processing time.

To summarize, it is difficult with the current API to tune the poll loop to avoid unexpected rebalances caused by processing overhead. To address this problem, we propose a way to limit the number of messages returned by the poll() call.

Public Interfaces

We add a new configuration setting *max.poll.records* to the KafkaConsumer API which sets an upper bound on the number of records returned in a single call to poll(). As before, poll() will return as soon as either any data is available or the passed timeout expires, but the consumer will restrict the size of the returned ConsumerRecords instance to the configured value of *max.poll.records*. The default setting (-1) will preserve the current behavior, which sets no upper bound on the number of records.

Proposed Changes

In this KIP, we propose to solve the problem above by giving users more control over the number of records returned in each call to poll(). In particular, we add a configuration parameter, *max.poll.records*, which can be set on the configuration map given to KafkaConsumer constructor. If the consumer fetches more records than the maximum provided in *max.poll.records*, then it will keep the additional records until the next call to poll(). As before, poll() will continue to send heartbeats in accordance with the configured heartbeat interval, and offset commits will use the position of the last offset returned to the user. The default configuration will preserve the current behavior, which places no upper bound on the number of messages returned in a call to poll(). Since the proposed change only affects configuration, users should not need to make any code changes to take advantage of it.

Below we discuss a couple implementation details.

Ensuring Fair Consumption

Since we are only returning a subset of the records that have been fetched, we must decide which ones to return. In a naive implementation, we might iterate through whatever collection the records are stored in until we've accumulated *max.poll.records* records and return these without further thought. In the worst case, this could leave some partitions indefinitely unconsumed since the consumer may fetch new data before the next call to poll(). The data would keep getting filled for the returned partitions and the unlucky ones stuck at the end would be starved.

Ideally, we would include records from each partition proportionally according to the number of assigned partitions. For example, suppose that the consumer has been assigned partitions A, B, and C. If *max.poll.records* is set to 300, then we would return 100 records from each partition. However, this is subject to the availability of the fetched data. If records are only available from A and B, then we would expect to return 150 records from each of them and none from C. In general, we can't make any guarantees about actual message delivery (Kafka itself does not return data proportionally in a fetch response for each requested partition), but we should give each partition a fair chance for consumption.

One way to achieve this would be to pull records from each partition in a round-robin fashion. Using the same example as above, we would first try to pull 100 messages from A, then 100 from B, and so on. We would have to keep track of which partition we left off at and begin there on the next call to avoid the starvation problem mentioned. This may require multiple passes over the partitions since not all of them will have data available. If no records were available from C, for example, then we would return to A. Alternatively, we could return after one pass with less records than are actually available, but this seems less than ideal.

For the initial implementation, we propose a simpler greedy approach. We pull as many records as possible from each partition in a round-robin fashion. In the same example, we would first try to pull all 300 records from A. If there was still space available, we would then pull whatever was left from B and so on. As before, we'd keep track of which partition we left off at so that the next iteration would begin there. This would not achieve the ideal balancing described above, but it still ensures that each partition gets consumed and requires only a single pass over the fetched records.

Prefetching

The KafkaConsumer implements a prefetching optimization to improve throughput. Before returning a set of records to the user in poll(), the consumer will initiate the next round of fetches in order to pipeline the fetching overhead and message processing. While the consumer is processing the current batch of records, the broker can handle the consumer's fetch requests (including blocking for new data if *fetch.min.bytes* is configured). The idea is to have data already available when the consumer finishes processing and invokes poll() again.

Since all available data is returned to the user in poll(), prefetching currently doesn't require a lot of work. In fact, the current logic is to prefetch on all partitions as long as there are no in-flight requests to the corresponding partition leaders. However, this becomes a little more complex with the addition of *max.poll.records* since the consumer may retain fetched data across multiple calls to poll(). It would not be desirable or necessary to initiate new fetches every time poll() was called if there was already enough data to satisfy the next call to poll(). We have considered the following heuristics to control prefetching:

1. Prefetch all partitions when the total number of retained records is less than *max.poll.records.*
2. Prefetch each partition when the number of retained records for that partition is less than *max.poll.records* divided by the current number of assigned partitions.

The simplest approach is the first one. Prefetching is skipped when there are enough records already available from any partition to satisfy the next call to poll(). When this number dips below *max.poll.records*, we fetch all partitions as in the current implementation. The only downside to this approach is that it could lead to some partitions going unconsumed for an extended amount of time when there is a large imbalance between the partition's respective message rates. For example, suppose that a consumer with max messages set to 1 fetches data from partitions A and B. If the returned fetch includes 1000 records from A and no records from B, the consumer will have to process all 1000 available records from A before fetching on partition B again.

The second approach attempts to address this problem by prefetching from each partition in accordance with its "share" of *max.poll.records*. Using the same example in the previous paragraph, the consumer would continue fetching on partition B while it works through the backlog of messages from partition A. The downside of this approach is that it could tend to cause more records to be retained. If a subsequent fetch for partition B returned 1000 records, then the consumer would retain roughly twice the number of messages as with the first heuristic.

In practice, we suspect that neither of the problems mentioned above are critical, so we opt for the simpler heuristic. Since the implementation does not affect the public API, it can be changed in the future without affecting users.

Compatibility, Deprecation, and Migration Plan

There will be no impact to existing users. The default value for *max.poll.records* will implement the current behavior of returning any data that is available with no upper limit on the number of records.

Rejected Alternatives

Introducing a second poll argument

An alternative way of limiting the number of messages would be to introduce a second argument to poll(), which controls the maximum number of records which will be returned. It would have the following advantages:

- It would allow the poll loop to dynamically change the max number of messages returned. This could be a convenient way to implement flow control algorithms which expand and contract the number of messages handled in each call to poll() according to the rate of handling.
- It would be more explicit. A counterargument is that there are many configuration options already. Avoiding yet another would not make that much difference.

However, it would also have the following disadvantages:

- The prefetching implementation could be more complex to implement since the consumer would not know how many messages would be consumed in the following call to poll().
- From an API perspective, we would have to either break compatibility or overload poll(). The first option is probably a non-starter and the second would arguably make the API more confusing.

It should also be mentioned that introducing a configuration parameter will not hinder us from introducing a second parameter to poll() in the future.

Adding a KafkaConsumer#heartbeat()/ping() method

One alternative that has been discussed is adding a heartbeat() API which sends a heartbeat and returns a flag (or throws an exception) to indicate that a rebalance is needed. This might change the typical poll() loop to something like the following:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(1000);
  for (ConsumerRecord<K, V> record : records){
    process(record);
    if (!consumer.heartbeat())
      break;
  }
  consumer.commitSync();
}
```

The problem with this approach is making it work in a reasonable way with offset commits. In the above example, breaking from the loop and committing before all messages have been processed will cause message loss. We can fix this by maintaining the offset map to commit explicitly, but it's unclear how to make it work with an auto-commit policy. In general, we felt that this added unneeded complexity to the API and that the same effect could be achieved in a safer way by setting *max.poll.records* to 1 and using the original poll() loop structure.