KIP-333: Add faster mode of rebalancing

Status

Current state: *Under Discussion*

Discussion thread: *Here*

JIRA: *KAFKA-7132*

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

Motivation

Currently, in Kafka, when a consumer falls out of a consumer group, it will restart processing from the last checkpointed offset. However, this design could result in a lag which some users could not afford to let happen. For example, lets say a consumer crashed at offset 100, with the last checkpointed offset being at 70. When the consumer recovers at a later time, the offset log being consumed has grown to offset 120. Consequently, there is a lag of 50 offsets (120 - 70). This is because the consumer restarted at 70, forcing it to start processing at a point far behind the end of the log. This lag is not acceptable in some computing applications (e.g. traffic control) who needs information by the millisecond, therefore, they will need a faster option for processing the offset lags that results from a crash. For example, after a consumer thread fails, the offsets after the crash point will have a longer latency before it is processed. Meanwhile, offsets before the crash point but after the last committed offset will be reprocessed to ensure that there is no data loss. We propose a new design to avoid the latency described by using two new threads instead of one. These two threads will process the offsets in such a way that it guarantees the user the data after the crash point with vastly reduced delay.

It has been noted that this is a relatively uncommon case among Kafka users. However, among streaming jobs which uses Kafka as a sink or source, this is a sought-after capability. Some streaming jobs such as Spark's Structured Streaming might have need for this proposal.

Public Interfaces

This new mode (name is `enable.parallel.rebalance`) will by default not be used by KafkaConsumer (e.g. will be by default equal to `false`). If the user chooses, he or she can activate this mode by setting the config's value to `true`.

With the introduction of this mode, the offset lag which results from a crash will be removed and any latency in terms of time that results would be negated (i.e. we continue processing as if the crash never happened). The user can use multiple threads of course to speed up processing of records to mitigate this lag, but they would only resolve the problem to an extent. The user does not have access to critical information in Kafka internals (for example, when a missed heartbeat triggers a LeaveGroupRequest). So the user are not in any position to exploit this information. Meanwhile, we can.

With the enabling of this mode, there will be an increased burden on a computer's processor cores because an extra consumer thread will be instantiated and run in parallel with the old consumer. In this manner, we could retrieve records at twice that of the current design. Note that when using this mode, offsets would no longer be given to the user in correct sequence (or order), but rather in a more haphazard manner. The mentality of this mode could be described as "give me all the offsets as fast as you can, I don't care about order." We will be able to guarantee at-least-once semantics, however the sequence which they come in will be thrown off. To illustrate, offsets could be sent to the user in this order: 1,2,3,6,4,7,5

A method called `childConsumerIsAlive()` will be introduced which returns a boolean value indicating if the secondary thread created has started or not. If true, then the thread is alive. This allows the user to check if a second thread is running.

Please note that the intension for this section of the KIP is only to given an overview of what will be done. Please take a look at the Design section below to get a better idea of how this mode operates.

Design

Here is the current design of a consumer and what happens when it recovers from a crash:

1) A consumer crashes at offset 100. Heartbeat thread detects that the foreground thread has stalled and decides to leave the consumer group.

2) When the consumer has recovered, on the next call to poll(), a JoinGroupRequest will be sent which marks the start of a rebalance. If the user were to call position() now, it will return the position of the last committed offset (offset 70) – which means that consumer will start consuming from this position.

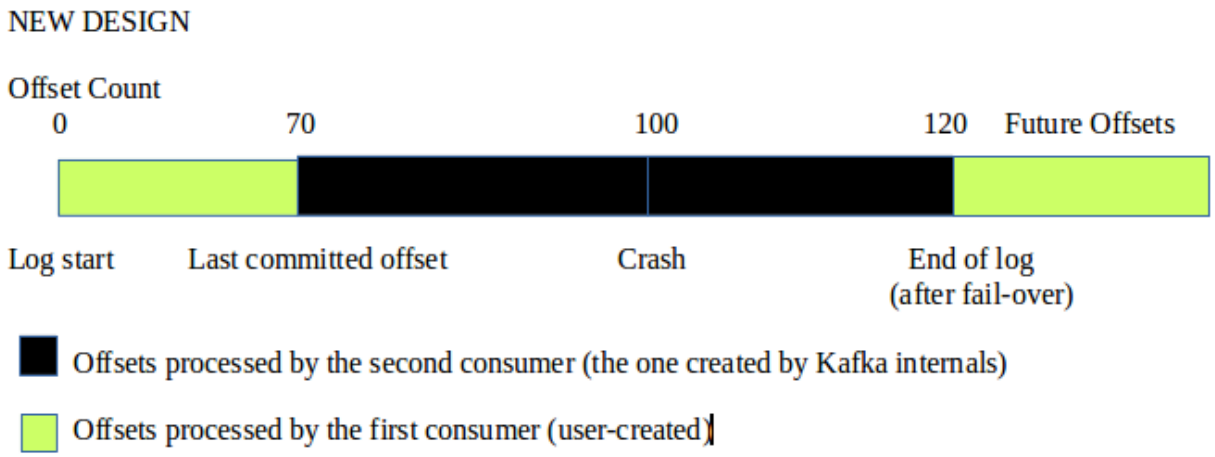
3) By this point, the log has grown to offset 120 (which means that we have a 50 offset lag - causing us to have a delay in terms of time).

With the proposed design, we could considerably speed things up by doing the following:

1) When the consumer that crashed recovers from the failure, it will seek to the end of the log (offset 120). Meanwhile, a new thread will be created to start processing the offset range that the first consumer skipped over (i.e. 70 - 120) and start consuming in parallel with the old consumer. Please note that with this design, the offset ranges being processed by both consumer threads will not overlap.

2) When the second consumer that was created had hit offset 120. We will not terminate this thread but rather let it go on standby for more failures from the preexisting thread.

3) Suppose that the first consumer crashed yet again, in this case, another lag could form as a result of this failure. We will repeat the same set of actions described previously, but without creating a new thread. In this case, we would have the first consumer again seek to the end of the log. Meanwhile, the second consumer will start processing the offset lag that results accordingly.



With this design, you could summarize the two consumer threads as follows:

1) The consumer created by the user processes the most recent data that has arrived, and will continue to do so even on the event of a crash

2) The consumer created by Kafka internals after a crash processes the offset lags that results from a crash, and will remain on standby once created in case another crash occurs

Regarding the effects on checkpointing with this design, instead of writing into one log. We will write into two commit logs instead of one. There is a couple of reasons supporting this option. For one thing, when writing into one commit log, there will be some concerns regarding thread-safety, particularly since the current design of KafkaConsumer allows only one thread to operate on it at a time. Two commit logs meanwhile could avoid that complication since threads will not be competing to write into one commit log. Another reason is that with this design, one consumer thread writing into a log could guarantee a sort of psuedo-ordering. (i.e. the offsets will be strictly increasing, not neccesarily being consecutive.) In comparison, two threads writing into one commit log will not be able to guarantee that. Just to make it clear, we will have these two threads described checkpointing in parallel as well.

Therefore, if we were to follow this design on checkpointing. We could have the following series of actions:

1) After a crash, the offset range 70 - 120 will be committed to a second commit log. Meanwhile, the first consumer thread (the one that crashed) will write offsets 120+ into the original commit log.

2) If there is yet another crash of the first consumer, then the lag that results will have its offsets be written to the second commit log. And you could probably guess where the remaining offsets would go based upon the policy described in the previous step. For example, if we were to crash again at offset 170 and recover at a point where the end of the log has grown to 190, with the last checkpointed offset being at 140. Then we will commit offsets 140 - 190 in the second commit log, and 190+ in the first.

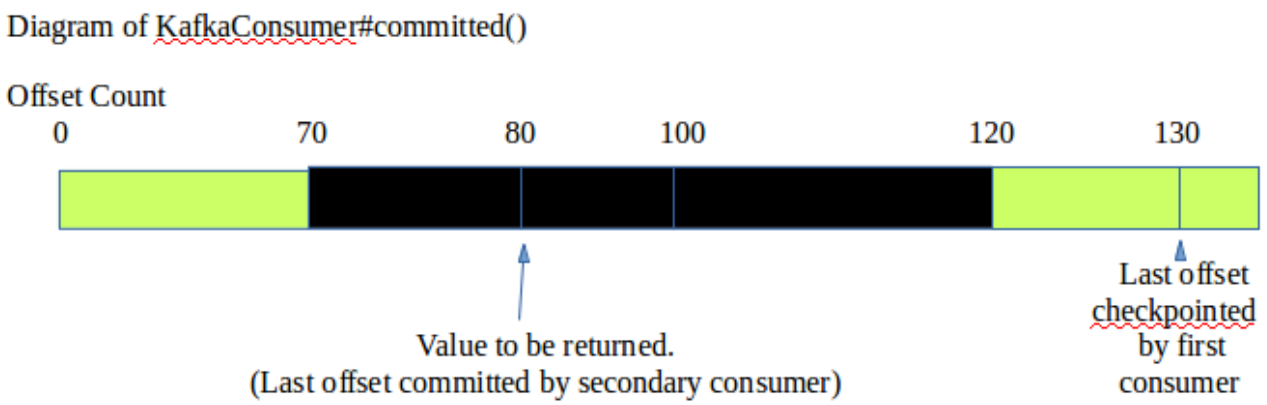
Please note that if auto commit is enabled, the two consumer threads will independently commit the offsets they have processed. Therefore, if one of the threads fail, then the other is not effected.

There is one corner case regarding crashes that had not been discussed yet, and that is if the secondary consumer (the one created by Kafka internals) would fail. In this situation, once it had recovered, it will resume processing from the last committed offset, unlike the first consumer which cannot do so. (This is because the information being processed is bounded, while the first consumer has to deal with a continuous influx of offsets). One piece of information that also needs to be stored in a commit log is the offset ranges themselves (i.e. 70 and 120 marks an offset range - we need to remember these two numbers so that we could tell when to stop retrieving records for the second consumer). To prevent us from losing track of which offset ranges to process, they need to be checkpointed as a token.

Effects on Current API And Behavior

poll() - when returning ConsumerRecords, if there are two consumer threads instead of one, we will have both of them return the records that they could obtain from their individual fetchers. In this manner, twice as many records could be returned. Some from the offset lag that results from a crash, and the offsets that had been recently added to the end of the log.

committed() - When returning the last checkpointed offset, rather than return the truly latest committed offset (i.e. the first consumer receiving the most recent data will probably have committed a offset for this partition), we will check if the second consumer (the one created by Kafka internals) has finished committing all offsets relating to this partition from the offset lags that result from a crash. If yes, then we will return the latest committed offset from the user-created consumer as planned. If no, then we will return the last offset that had been committed by the second consumer. This guarantees that any offset before the one returned by KafkaConsumer#committed has also been checkpointed as well.



position() - will call the user-created consumer's position, the second consumer (created by Kafka internals) will not take any part in this call

commitSync() and commitAsync() - We will split the offsets that wants to be committed into two groups. One group has offsets that the second consumer is processing (i.e. if an offset to be committed is within one of the offset lags created by a crash, then it will be added to this group). This group will be committed by the second consumer since the offsets in the group were processed by the second consumer as well. Meanwhile, another group (which contains then remaining offsets) will be committed by the first consumer.

subscribe(), assign(), and unsubscribe() - similar to position(), will not call second consumer's methods.

beginningOffsets(), endOffsets() - Will check both consumer threads for the earliest/latest offsets that they have processed and will return them.

offsetsForTimes() - Will not be changed since the partitions (which is the input argument for this method) and their respective offsets does not have to be assigned to any specific consumer.

All remaining methods in KafkaConsumer will not be impacted too heavily by this mode. There will be some minor changes. For example, when we call pause() or resume(), both consumer threads will stop or restart processing on partitions accordingly. While close() will cause both consumers to shutdown.

Compatibility, Deprecation, and Migration Plan

No compatibility issues - - only changing the internals of how some of the methods work.

Rejected Alternatives

It has been considered whether or not to allow this mode to support ordering guarantees. After some thought, it was rejected since the implicit costs that comes with it is too high.

For there to be ordering guarantees, it will require the processes involved to either wait for the latest uncommitted offsets to be checkpointed (i.e. all offsets are stored in the same log), or write into distinct logs which could be queried. The former option is too costly in terms of latency, particularly if there is another crash. The latter option is better in terms of latency in that we don't have to wait for other offsets to be fed into the commit log before checkpointing. However, such an approach could lead to an explosion in the number of topics that store the committed offsets. In such a scenario, if there were several crashes within a certain period of time, the number of topics would skyrocket. This is not acceptable.

As for polling(), ordering guarantees could not be supported either, particularly since the process which is consuming the earliest uncheckpointed offsets could be the only thread which returns it results. The other thread(s) involved will not be able to return anything until the first process finished with those offsets.