KIP-408: Add Asynchronous Processing To Kafka Streams

Status

Current state: *Under Discussion*

Discussion thread: *Discussion Link*

JIRA: *KAFKA-6989*

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

Motivation

Currently, in Kafka Streams, a single thread maximum is allowed to process a task which could result in a performance bottleneck. With the current model, under the condition of a fatal error, a new retry topic will be created/used in which failed records will be sent. Incoming records could not be processed until the failed records in the retry topic are fully reprocessed to guarantee ordering. This could result in a lag because we are essentially backtracking to make sure all data has been sent. New data that would have been processed if a failure had not occurred will have to wait while the thread paused on previous records.

Present Kafka Design Philosophy

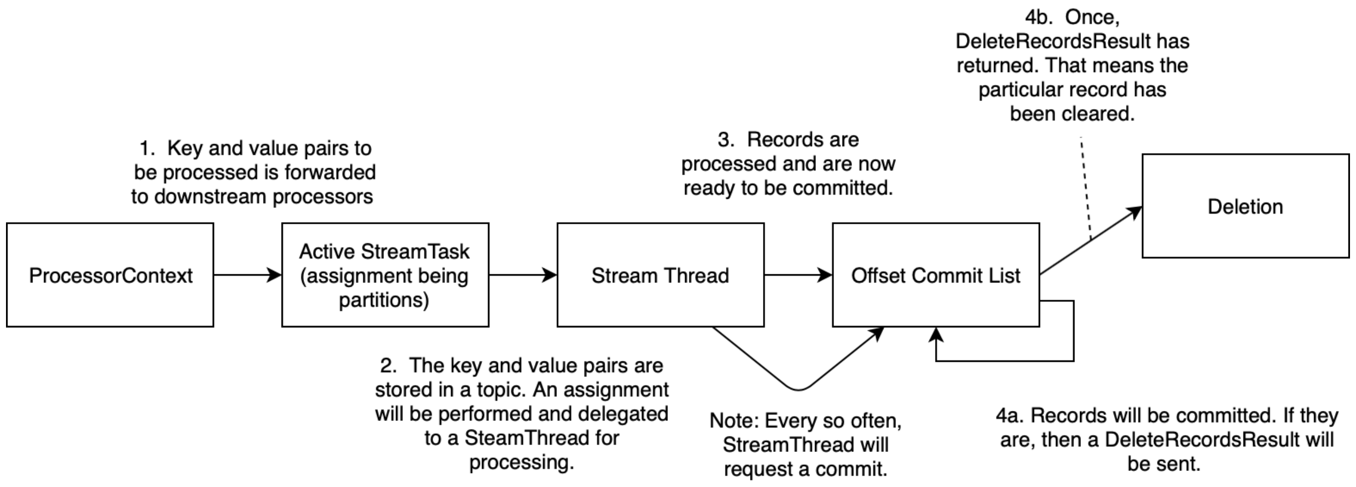
Before we dive into how we could change the preexisting codebase, it would be helpful to clarify on some of the details regarding the current Kafka Streams design. Right now, threads are independent of one another–that is, each thread is in charge of its own stream task. There is no cooperation between threads. And as a result, they are not processing the same partition, but instead distinct ones. For example, one thread processes partition1 while another processes partition2. These two threads does not attempt to access the other topic because such knowledge is not given to them. In this manner, the two threads are isolated from one another.

However, this design could be problematic under certain conditions. When failed records are moved to a retry topic, the current thread has no way of passing it off to another thread and say "here, you could do this work for me so it doesn't hold up the line." Generally, a API which saves the user most implementation details seems to be favorable, therefore we will likely have to do most of the heavy lifting ourselves (that is implement the multithreading and the extra asynchronous processes ourselves, kinda like Samza).

Currently, when a new KafkaStreams applications is initialized, the user would have a config available to them which defines the number of threads KafkaStreams will use (`num-stream-threads`). What will happen is that N `StreamThread` instances would be created where N = `num-stream-threads`. However, if N is greater than the number of tasks, then some threads will be held in reserve and will idle unless some other thread fails, in which case, they will be brought online. By current structure, each task will have at maximum one thread processing it.

Please note that any new additions made by this KIP will probably affect only internals. The current methods found in Processor API and their behavior would most likely not be changed.

Processor API Structure


4b. Once, DeleteRecordsResult has returned. That means the particular record has been cleared.
1. Key and value pairs to be processed is forwarded to downstream processors
3. Records are processed and are now ready to be committed.
Deletion
ProcessorContext
Active StreamTask (assignment being partitions)
Stream Thread
Offset Commit List
2. The key and value pairs are stored in a topic. An assignment will be performed and delegated to a SteamThread for processing.
Note: Every so often, StreamThread will request a commit.
4a. Records will be committed. If they are, then a DeleteRecordsResult will be sent.

Above, we could see a simplified diagram of how KafkaStreams implements the Processor API. It should be noted that one StreamThread can process records from multiple StreamTasks at once. But this is not applicable in reverse. A StreamTask could not be sending records to multiple StreamThreads. This is a major bottleneck and we would need to work to fix this.

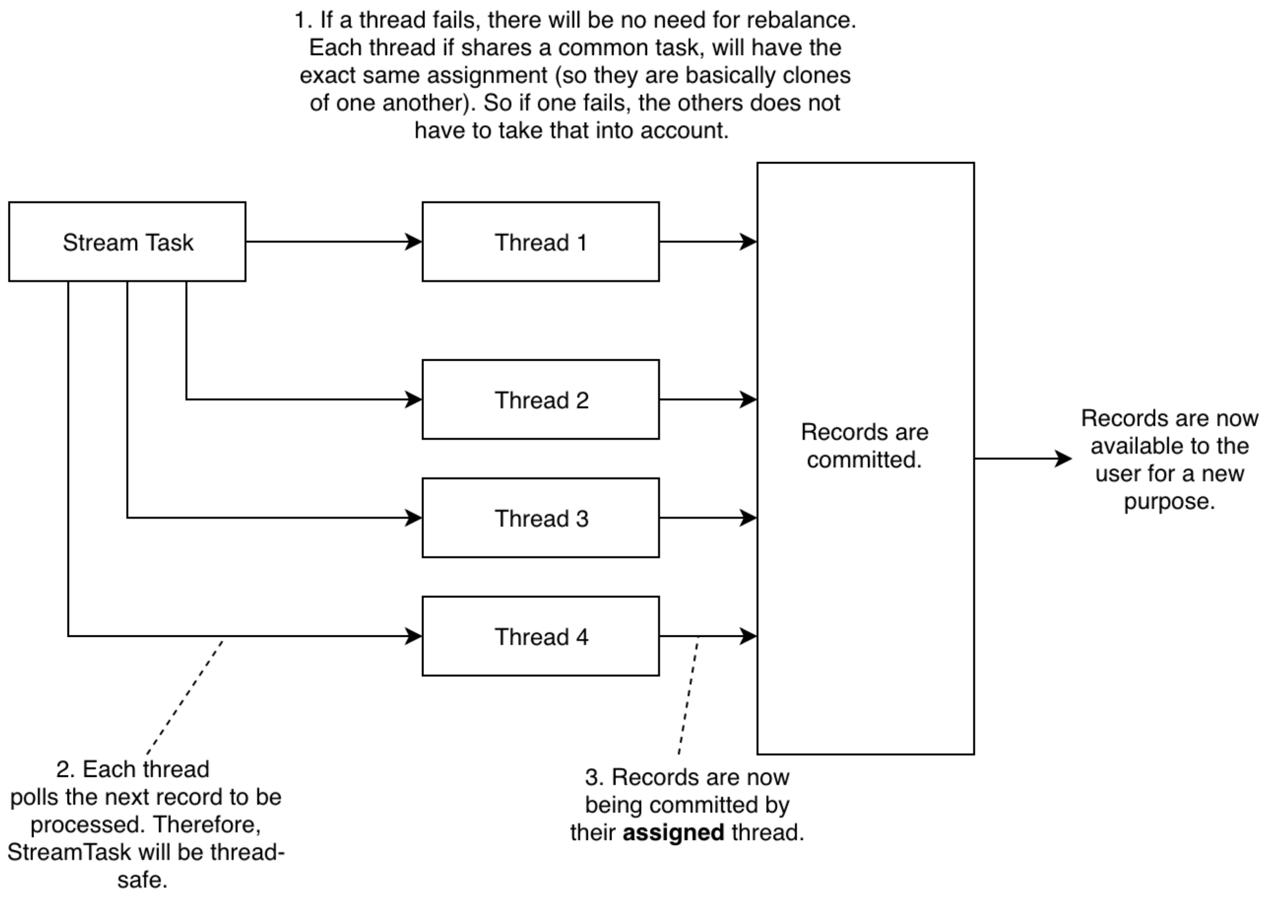
Public Interfaces

We have a couple of choices available to us as mentioned in the ticket discussion (*KAFKA-6989*). Recall that in asynchronous methods, we do not need inputs from a previous function to start it. All we need is the signal. However, when offsets have been committed or processed in an asynchronized manner, we will need to determine the behavior that occurs after the conclusion of the process. It would also be wise to remember that asynchronous processing does **not** necessarily require an extra thread, but in this case, it is necessary. When offsets are committed/processed, we should consider the following:

1. Ordering: As noted in the JIRA chat, Samza has judged it to be impossible for the records to be returned in its original sequence by their implementation.
2. Exactly-Once: In exactly-once semantics, each offsets are returned once. This will hard if multiple threads are active (i.e. more than one thread is calling the same Kafka Streams instance).
3. Latency and Resilience: Whenever we attempt to retry processing, as mentioned in the JIRA ticket, it could end up as a performance bottleneck because we are effectively "stopping the world" by pausing on a single record. An option to avoid this is to allow a second thread to handle these failed records while we continue to process incoming metadata. However, exactly once and ordering would not be supported under these conditions.

To speed things up, we should consider how to use multiple asynchronous threads to process from one StreamTask (i.e. receive records from the StreamTask to be processed).

1. Positive sides: Failure handling is better now in that multiple threads are on the job. While a secondary thread could possibly take care of the failed metadata, the primary thread could move on processing new ones. Now the records to be processed are being split between threads, we could process them faster.
2. Negative sides: Ordering is now harder to guarantee, and exactly-once is impossible because we have no way of knowing which records has been returned since asynchronous threads have no way of communicating between one another.


1. If a thread fails, there will be no need for rebalance. Each thread if shares a common task, will have the exact same assignment (so they are basically clones of one another). So if one fails, the others does not have to take that into account.
Stream Task
Thread 1
Thread 2
Thread 3
Thread 4
Records are committed.
Records are now available to the user for a new purpose.
2. Each thread polls the next record to be processed. Therefore, StreamTask will be thread-safe.
3. Records are now being committed by their **assigned** thread.

This would be a flow diagram of how processing records would work if there is multiple threads processing records at once asynchronously. For ordering to be guaranteed, here is what will happen:

1. A Map with (Key, Value) = (TopicPartition, AtomicInteger) will be created. Any TopicPartition, when first inserted into the map, will have its AtomicInteger value set to zero. This variable will be used to denote the current offset which we have most recently committed.
2. An offset for a particular TopicPartition will not be committed unless the AtomicInteger value indicates it is eligible for a commit operation. Once the commit operation is complete, the AtomicInteger value will be incremented for that particular partition to tell other StreamThreads that the next record could now be sent to the user.
3. This AtomicInteger reference will be shared across all StreamThreads. So that each time an AtomicInteger value is updated, other threads will be notified of this update.

In this manner, we will be able to commit records in sequence.

Proposed Changes

A new config will be added (like `num.threads.per.task`) which will define how many threads will be used per task. By default, it will be set to one, which means that the current behavior of Processor API will be replicated. However, if it is more than one, two or more StreamThreads, for instance, will be processing at the same time from a task. Ordering is guaranteed, therefore, the user does not have to deal with inexplicable out-of-order issues. There will be an added benefit of faster processing and commit speeds for that particular task.

Note: if the total number of threads is not divisible by the `num.threads.per.task` then a task will probably have the remainder of the threads assigned to it after division. An `IllegalArgumentException` will be thrown if `num.stream.threads < num.threads.per.task.`

Compatibility, Deprecation, and Migration Plan

There are no considerations for deprecation, particularly since we are adding a new capability, not upgrading a preexisting one.

Impacts and Steps for Implementation

There are a series of ramifications that would result from this KIP that we would need to take into account. For starters, the metrics for KafkaStreams will need to be updated such that it could output the states of multiple threads if they are working in tandem on the same Kafka Streams application (but this will come later once we have laid the groundwork for the other methods).

Rejected Alternatives

N/A at the moment.