KIP-287: Support partition and processor expansion for stateful processing jobs

- Status
- Motivation
- Goals
- Public Interface
- Proposed Changes
    - Support partition expansion
    - Support processor expansion
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *here*

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

Motivation

KIP-253 ensures that messages with the same key from the same producer can always be delivered in the order that they are produced. However, for stateful stream processing jobs, after we change the number of partitions or the number of processors, the messages with the same key may go to a different processor that does not have the state necessary to process this message. This may cause stateful stream processing job to output wrong result.

In this KIP we propose a solution that allows user to expand number of partitions of input topic as well as the number of processors of stateful processing jobs while still ensuring output correctness.

Note: In this KIP, we use the terminology "processor" to indicate a combination of consumer, local state store and user-specified processing logic.

Goals

In addition to ensuring correctness, we also aim to achieve the following goals to optimize the efficiency of this KIP.

1) When we expand partitions of input topics, we should not change the key -> processor mapping, so that no state has to be moved between the existing processors. This in turn would guarantee correctness because KIP-253 ensures that messages with the same key can be delivered in the order that they are produced.

2) When we expand processors of a given stateful processing job, some key will be assigned to the newly-added processors. And it would be necessary to copy the state for the corresponding keys to these processors before these processors can consume messages and generate output. However, we should not have to copy state between existing processors.

3) In order to keep processing new messages in near real-time when we are copying state to the newly-added processor, the newly-added processor should only start to consume "new" messages and generate output "after" its local state is caught up. Before its local state is caught up, the existing processor should continue to process new messages and generate output.

4) In order to avoid disrupting existing use-case and keep backward compatibility, the 3-tuple (topic, partition, offset) should uniquely identify the same message before and after the expansion. And a given message should be consumed exactly once by bootstrapping consumers after the partition expansion. This goal suggests that we should not delete existing message or copy any message in the topic.

Public Interface

TODO

Proposed Changes

Support partition expansion

Here we describe how to support partition expansion while keeping the same key -> processor mapping.

Here are the requirements. We skip the detail for now (e.g. how to remember the initial number of consumers in the consumer group).

1) Stateful processing job starts with the same number of processors as the initial number of partitions of the input topics

2) At any given time the number of partitions of the input topic >= the number of processors of the processing job. In other words, we always expand partitions of the input topic before expanding processors of the processing job.

3) The processing job remembers the initial number of processors in the job. This can be supported in core Kafka by remembering the initial number of consumers in the consumer group.

4) The partition -> processor mapping is determined using linear hashing. This is where we need the initial number of processors in the job.

It can be **proven** that the above requirements would keep the same key -> processor mapping regardless of how we change the partition number of the input topics, as long as we don't change the number of processors of the processing job.

We skip the proof here. Reader can thinking about how the key -> processor mapping changes in the following example:

- Input topic initially has 10 partitions and processing job initially has 10 processors.

- Expand partition of the input topic to 15.

- Expand processors of the processing job to 15.

- Expand partition of the input topic to 18.

Support processor expansion

Here we provide the high level idea of how this can be supported. Most of the logic is expected to be implemented in the stream processing layer.

Note: It is required that, at any given time, the number of partitions of the input topic >= the number of processors of the processing job. In other words, we should always expand partitions of the input topic before expanding processors of the processing job.

When a new processor is added to the processing job, the processor should be able to know the following information:

- The set S1 of partitions that the new processor should be consuming after its local state is caught up.

- The latest offsets T1 of partitions in the set S1 that have been consumed by the existing processors in the job. This information should be periodically retrieved over time.

- The set S2 of partitions that may contain messages with the same key as the messages in S1. S2 includes all partitions in S1. Set S2 can be determined using the logic added in KIP-253.

- The latest offsets T2 of partitions in the set S2 that has been used to generate the local state store.

Initally offsets T2 will be 0 for all partitions in the set S2 because the local state store is empty. Offsets T1 will keep growing as the existing processors keep consuming messages from partitions in the set S1. The new processor should keep reading messages from partitions in S2 to re-build local state until offsets T2 have "almost" caught up with offsets T1 of the existing processors for all partitions in S1. Note that this follows the design in KIP-253 such that, before any message can be read by the new processor from the newly-added partitions in S1, all prior messages with the same key (as any message in partitions S1) must have been consumed in order.

Also note that, while the new processor is catching up, the existing procesor will still be consuming all new messages of all partitions of the input topic and no rebalance has been triggered yet. And new processor uses the messages to re-build local state without generating any output.

After offsets T2 have "almost" caught up with offsets T1, we should trigger rebalance to re-assign the partitions in S1 from the existing procesor to the new processor. The existing processor should commit offset, generate output and stop consuming messages from partitions in S1. The new processor should first consume messages from offset T2 to offset T1 to finish re-building the local state. Then it can consume messages starting from offsets T1 and genreate output for these messages.

Compatibility, Deprecation, and Migration Plan

TODO

Rejected Alternatives

TODO