KIP-311: Async processing with dynamic scheduling in Kafka Streams

- Status
- Motivation
- Public Interfaces
- Proposed Changes
  - Stage 1: Issue
  - Stage 2: Execute
  - Stage 3: Store & commit
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives
  - Lightweight approach

Status

Current state: IDLE - THIS IS KIP IS ABANDONED, if you're interested in the subject, feel free to take it over. (I intended to work on this but priorities shifted and it's now rather obvious that I can't do it, sorry)

Discussion thread: *here [TBD: update]*

JIRA: *here*

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

Motivation

Today Kafka Streams use a single-thread per task architecture to achieve embarrassing parallelism and good isolation. However there are a couple scenarios where async processing may be preferable:

1) External resource access or heavy IOs with high-latency. Suppose you need to access a remote REST api, read / write to an external store, or do a heavy disk IO operation that may result in high latency. Current threading model would block any other records before this record's done, waiting on the remote call / IO to finish.

2) Robust failure handling with retries. Imagine the app-level processing of a (non-corrupted) record fails (e.g. the user attempted to do a RPC to an external system, and this call failed), and failed records are moved into a separate "retry" topic. How can you process such failed records in a scalable way? For example, imagine you need to implement a retry policy such as "retry with exponential backoff". Here, you have the problem that 1. you can't really pause processing a single record because this will pause the processing of the full stream (bottleneck!) and 2. there is no straight-forward way to "sort" failed records based on their "next retry time" (think: priority queue).

3) Delayed processing. One use case is delaying re-processing (e.g. "delay re-processing this event for 5 minutes") as mentioned in 2), another is for implementing a scheduler: e.g. do some additional operations later based on this processed record. based on Zalando Dublin, for example, are implementing a distributed web crawler. Note that although this feature can be handled in punctuation, it is not well aligned with our current offset committing behavior, which always advance the offset once the record has been done traversing the topology.

Public Interfaces

Briefly list any new interfaces that will be introduced as part of this proposal or any existing interfaces that will be removed or changed. The purpose of this section is to concisely call out the public contract that will come along with this feature.

A public interface is any change to the following:

- *Binary log format*
- *The network protocol and api behavior*
- Any class in the public packages under clientsConfiguration, especially client configuration
  - org/apache/kafka/common/serialization
  - org/apache/kafka/common
  - org/apache/kafka/common/errors
  - org/apache/kafka/clients/producer
  - org/apache/kafka/clients/consumer (eventually, once stable)
- *Monitoring*
- *Command line tools and arguments*
- *Anything else that will likely break existing users in some way when they upgrade*

Proposed Changes

We should add a mechanism for out-of-order execution with in-order commit - not unlike the dynamic scheduling algorithm/ architecture invented by Tomasulo in the '60s. In that sense,

- "Instruction units" are the ***Source Processors*** - they will produce an input stream of data and forward it to the downstreams processors. In this analogy, the stream tasks are "instructions" to be executed by the processor nodes ("execution units")
- "Storage units" are the ***Sink processors*** - they will take the results and write them to the destination topics. They will also commit offsets, where appropriate (in-order)
- "Reservation stations" - this is a new concept that we'll need to introduce; it is partially handled by the cache. The KafkaStreams app needs to have a well-defined number of reservation stations, that store intermediate results, keep track of which results are ready, solve WAW hazards (e.g. when writing to a changelog topic, one may simply cancel an earlier operation) and potentially RAW hazards (forward internally to processors before producing explicitly to the output topic). Not sure WAR hazards apply.
- "Functional Unit" - the AsyncProcessingNode
- The scheduler in the KafkaStream app plays the role of the "common data bus"; i.e. it makes sure that the communication between all reservation stations happens, but also checks that I/O is complete, retires completed/canceled operations (clears reservation stations).

With this proposal, a message goes through a StreamTask in three stages:

Stage 1: Issue

- wait until there's an empty RS
- Retrieve next record ("instruction") from the source processors, fill in the RS

Stage 2: Execute

- wait until source data is available (async processing completed in prior processor(s))
- start async processing
- mark RS as "ready" on callback (when I/O is finished)

Stage 3: Store & commit

- (this happens in the sink processors) Write result to the output topic,
- Mark input partition/offset "complete"
- If there's a prefix of "completed" or "cancelled" offsets commit the last offset for partition & discard the prefix.

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users?*
- *If we are changing behavior how will we phase out the older behavior?*
- *If we need special migration tools, describe them here.*
- *When will we remove the existing behavior?*

Rejected Alternatives

This is not really "rejected" - in fact, it's what I initially considered to be the best approach; in the meanwhile I just thought that the "Reservation approach" would be an elegant way to solve this in the general case with little headache for the users, and with enough generality that it can be customized/ configure to handle the most advanced usecases.

Lightweight approach

Just make the commit() mechanism more customizable to users for them to implement multi-threading processing themselves: users can always do async processing in the Processor API by spawning a thread-poll, e.g. but the key is that the offset to be committed should be only advanced with such async processing is done. This is a light-weight approach: we provide all the pieces and tools, and users stack them up to build their own LEGOs.