Problem Summary

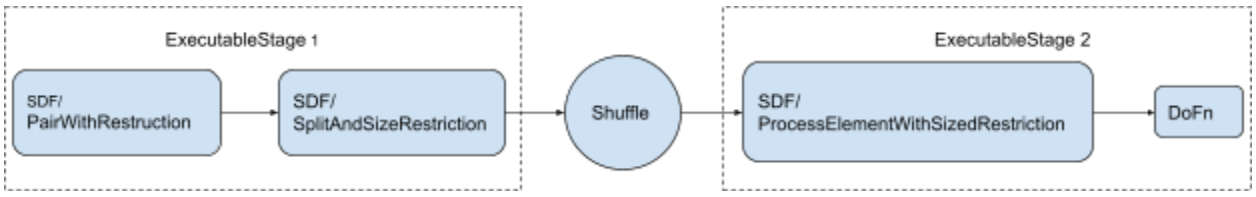
Beam Splittable DoFn is able to self-checkpoint on each element and send back residuals via ProcessBundleResponse.DelayedBundleApplication. The runner should reschedule these residuals later based on the provided resumeDelay.
Currently Dataflow runner has supported executing residuals for both batch and streaming by using runner v2. We want to have Flink to support such functionality in order to fully support executing splittable DoFn.

Beam on Flink

When FlinkPipelineRunner takes Beam pipeline proto, it will first expand splittable DoFn(if any) into SDF/PairWithRestriction -> SDF/SplitAndSizeRestriction -> SDF/ProcessElementAndSizeRestriction by using SplittableParDoExpander. Then it replaces known PTransforms with native implementation and fuses the pipeline with GreedyPipelineFuser into several ExecutableStage. Finally the FlinkPipelineRunner translates the fused pipeline into Flink Operator and Function, where we have ExecutableStageDoFnOperator to execute one fused Beam sub-graph in streaming execution and FlinkExecutableStageFunction in batch processing by talking to Beam SDK harness over fnapi.
SDF/ProcessElementAndSizeRestriction is the DoFn that can produce DelayedBundleApplication and the FlinkPipelineRunner should feed residuals back to this DoFn.

Potential Approaches

In order to feed residuals back to SDF/ProcessElementAndSizeRestriction, SDF/ProcessElementAndSizeRestriction needs to be the root transform of one ExecutableStage. That requires to insert a fusion break between SDF/SplitAndSizeRestriction and SDF/ProcessElementAndSizeRestriction so that the graph looks like:

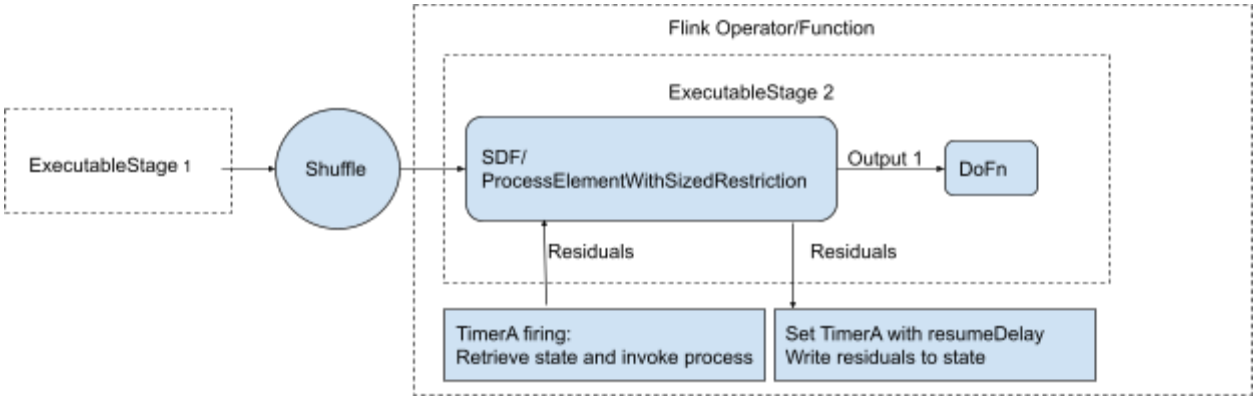


Inserting fusion break has been done via GreedyPCollectionFusers.

State and Timer can be used to reschedule residuals by setting the Timer with resumeDelay and writing residuals to the state. When the timer is fired, we can retrieve the state and feed residuals back to the ExecutableStage. There are 2 potential ways to feed them back.

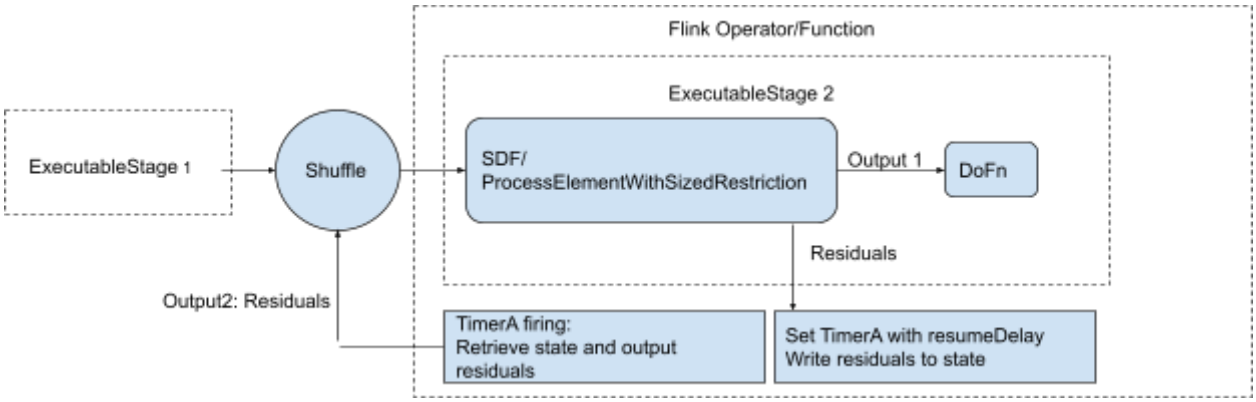
Approach 1: Feed residuals back to itself

When the timer is fired, the Flink Operator/Function invokes the ExecutableStage with residuals directly.



Approach 2: Feed residuals back to the Shuffle

Instead of invoking the ExecutableStage directly inside the Flink Operator/Function, the ExecutableStage can also output the residuals to the previous Shuffle step. The assumption is that the Flink is able to run with a cyclic graph.



This is more about extendable support for dynamic split as well. The common part for dynamic split and self-checkpoint is rescheduling residuals. Different from self-checkpoint, we want to redistribute the work among different workers when dynamic split happens. That requires the ExcutableStage can output residuals back to the Shuffle step.

There are some concerns around this cycle approach. First, not all runners support executing the cyclic graph, thus this approach will be limited to Flink only. Besides, there are also some limitations around cyclic graphs. Please refer to this thread for more detailed discussion around this.

Decision

We decided to go with Approach 1 for simplicity and possibility to have the shared Java implementation for OSS runners.

Implementation Details(Approach 1)

Java Runner Shared Library

StateAndTimerBundleCheckpointHandler
A StateAndTimerBundleCheckpointHandler is designed for all Java runners to handle SDF initiated checkpoint as long as the runner supports Timer and State. A timer is set to reschedule these residuals and the delayed residuals will be written to the state.
To construct a StateAndTimerBundleCheckpointHander, 4 objects are needed:

- A TimerInternalsFactory, which is used to create a TimerInternals for each DelayedBundleApplication.BundleApplication.element.
- A StateInternalsFactory, which is used to create a SateInternals for each DelayedBundleApplication.BundleApplication.element.
- A window coder for the current bundle. This window coder is used to create StateNamespace for both setting timer and writing state.
- An input coder for the current bundle. This coder is used to decode the residual element into a windowed value, which is the key for TimerInternalsFactory.forKey() and StateInternalsFactory.forKey().

Inside onCheckpoint(ProcessBundleResponse), several things happen for each DelayedBundleApplication:

- StateAndTimerBundleCheckpointHandler creates a unique id, which is used as the id for both the timer and state.
- A TimerInternals and a StateInternals are created for the decoded residual value.
- A processing time timer is set where:
    - timestamp = now() + DelayedBundleApplication.RequestedTimeDelay
    - outputTimestamp = min(DelayedBundleApplication.BundleApplication.outputWatermarsMap) If there is no output watermark, the outputTimestamp will be the MIN_TIMESTAMP
- A ValueState is written where:
    - The StateNamespcate is the same as the timer
    - The id is the same as the timer
    - The coder is the input coder.

StageBundleFactory
The BundleCheckpointHandler is exposed to the StageBundleFactory.getBundle() API just as other handlers.

Flink Changes

Batch translation and execution

We use InMemoryTimerIntenrals and InMemoryStateInternals for any given keys in Batch, which makes the batch case much simpler.

When the FlinkIBatchPortablePipelineTranslator translating ExecutableStage, the input coder is exposed to the FlinkExecutableStageFunction. The FlinkExecutableStageFunction keeps a StateAndTimerCheckpointHandler if there is a splittable DoFn and requests processing the bundle with the checkpoint handler.
When a bundle finishes processing, the FlinkExecutableStageFunction will check whether there are any pending SDF timers. If so, the FlinkExecutableStageFunction retrieves the delayed residual elements from state and invoke a new bundle to process these elements.

Streaming translation and execution

Different from batch processing, we use FlinkTimerInternals and FlinkStateInternals to work with states and timers.

SdfByteBufferKeySelector
For a splittable DoFn, we already know that the format of the input should be KV<KV<element, KV<restriction, watermarkState>>, size>, where we will use the element as the key for Flink to use the timers and keyed state. The SdfByteBufferKeySelector is designed to extract such a key from one given input of a splittable DoFn. The input DataStream will be keyed by this key selector and the ExecutableStatgeDoFnOpeator will use the key selector to set the key for the state backend.

SdfTimerInternals/SdfStateInternals and SdfTimerInternalsFactory/SdfStateInternalsFactory
FlinkTimerInternals and FlinkStateInternals require the caller to set the key for the Flink state backend and grab the lock before setting timers/states, thus a wrapper is needed for both. SdfTimerInternals wraps the FlinkTimerInternals to deal with the key and lock, and SdfTimerInternalsFactory is for creating a SdfTimerInternals with one given key. Similar to timer, we also have SdfStateInternals and SdfStateInternalsFactory for the same reason.

During execution, the ExecutableStageDoFnOperator requests processing a bundle with the StateAndTimerBundleCheckpointHandler. When an SDF timer is set, the FlinkTimerInternals sets the watermarkHold by using the outputTimestamp from the SDF timer. We rely on Flink to fire any SDF timer. When an SDF timer is firing, the ExecutableStageDoFnOperator will first remove the watermarkHold and then retrieve states to invoke processElement().