Pipeline Drain

Shared with Apache Beam Community

Reuven Lax <relax@google.com>

Summary

Users running continuous pipelines over unbounded input data need a way to safely drain data out of the pipeline in preparation for a restart without any data loss.

This document proposes a new pipeline action called Drain. Drain can be implemented by runners by manipulating the watermark of the pipeline. This document also proposed some extensions to the API to allow users to perform specific actions when a pipeline is drained.

Problem Definition

Apache Beam provides a unified processing model for both bounded and unbounded input data. Users processing unbounded data do so via a continuous, always-running pipeline[1]. When the time comes to stop this pipeline (possibly to launch a new one), users want to ensure that no data buffered in the old pipeline (e.g. in a window buffer waiting for the watermark to advance) gets lost. Due to the nature of Beam's windowing model, there's never a point in time where it's safe to simply stop the pipeline - there might always be records still in the pipeline. We need a way to drain all buffered data out of the pipeline before stopping it to ensure data is not lost.

Note that we do intend to also propose[2] a complementary feature that allows updating a running pipeline in place, as long as the new pipeline is compatible. This is a complementary feature, with somewhat different use cases than Drain.

Drain

We propose the following definition for drain. Drain is an external operation on an unbounded source[3] that converts it to a bounded source. Once all unbounded sources have been converted to bounded sources, the pipeline will finish just like batch pipelines do. Finished bounded sources are at the end of the global window, so all buffered windows will be processed, and no elements will be left behind.

---

[1] Even a micro-batch runner is "always running" for the purposes of this document
[2] Proposal upcoming soon.
[3] A similar definition will hold for splittable-dofn sources.

Runners implement these semantics in terms of watermarks. Once a given unbounded source is drained, its watermark is advanced to infinity[4] and the source stops producing elements. This will cause all downstream timers and windows to finish, draining them out of the pipeline. The watermark advancement will cascade down the pipeline as steps complete. Once the watermarks for all steps have advanced to infinity, the pipeline is drained and can be stopped; the runner should automatically stop the pipeline once all watermarks have advanced to infinity.

We explicitly exclude bounded sources from Drain. Often when a pipeline is processing both bounded and unbounded sources, the bounded source is used as a side input into steps processing the unbounded source. It may not be possible to correctly process *any* elements from the unbounded source without a complete side input, so it doesn't make sense to cut off reading from the bounded source. There may be good use cases to cutoff reads of bounded sources, however that is a different feature and a different proposal (and one that applies to batch pipelines as well).

While we've defined Drain as an operation on a source, how this operation is exposed to the user runner determined. Many runners may decide to expose this as a pipeline-level operation, in which case draining a pipeline means draining all unbounded sources in the pipeline.

Note that this definition of drain might result in a pipeline producing incomplete windows. For example, consider a pipeline that uses ten-minute fixed windows. If Drain is called at 12:02pm (watermark time), then the 12:00-12:10 window will be output containing only elements from 12:00-12:02. However, this is is simply the inverse of pipeline behavior when a pipeline first starts! If the pipeline first starts reading data at 12:08, then it will output the 12:00-12:10 window containing only elements from 12:08-12:10. We believe that many use cases will be fine with incomplete windows when a pipeline is drained.

Google Cloud Dataflow

This proposal is inspired by the Drain feature long supported by Google Cloud Dataflow. However the Dataflow Drain functionality is fairly basic, and does not allow users to perform any actions when a pipeline is drained. It also does not provide a way of draining keyed state, and does not work well with many types of sources. The goal of this proposal is not to copy Dataflow's feature, rather it is to propose a more-developed drain for Beam that all runners can benefit from.

PaneInfo

Remember that draining a pipeline can result in outputting incomplete windows. While many use cases are fine with this, there are times when a pipeline needs to know that the window it is

[4] Where infinity is defined to be GlobalWindow.MAX_TIMESTAMP.

outputting is incomplete. Following are two use cases, both loosely based on real scenarios encountered by users:

Use Case 1: Detecting end of videos

X has a website that hosts a video player. The video player publishes events when a video is started, stopped, etc., and during play it publishes an event once a second to indicate that play is still in progress. Since users can navigate away from the player at any time, the player cannot always publish a finished-playing event. All events are tagged with a play_id, for that play session. X wants to use Beam to determine when such videos have finished, and hits upon Session windows as a good approach. If more than 5 minutes go by without an event published for a given play_id, assume that the video is finished and publish the finish time.

The problem with Drain is that all in-progress session windows will complete, causing this pipeline to publish bogus finish times for in-progress plays. X would like to know that a window is ending due to a Drain, and mark the output as tentative.

Use Case 2: Rereading Output

Y has a pipeline that calculates 10-minute windowed sums of users. The code[5] for the pipeline is as follows.

```
p.apply(read())
 .apply(Window.into(FixedWindows.of(Duration.standardMinutes(10))))
 .apply(WithKeys((UserEvent e) -> e.getUserId()))
 .apply(Count.perKey())
 .apply(WriteOutput());
```

In the case of Drain, Y does not want to write incomplete windows to the main output. Instead Y would like to write the partial sums out to a different location and read that location into the new pipeline.

Proposal

We propose exposing this to the user via PaneInfo. PaneInfo already contains isFirst() and isLast() accessors, we will add an isDrain() accessor while will indicate that the pane is firing due to drain.

[5] pseudocode

Use case 2 will use this as follows:

```
PCollection<KV<String, Integer>> mainInputCounts = p.apply(read())
   .apply(Window.into(FixedWindows.of(Duration.standardMinutes(10))))
   .apply(WithKeys((UserEvent e) -> e.getUserId()))
   .apply(Count.perKey());

PCollection<KV<String, Integer>> incompleteCounts = p.apply(readIncompleteAggregations())
   .apply(Window.into(FixedWindows.of(Duration.standardMinutes(10))))
   .apply(Keys.create())
   .apply(Count.perKey());

PCollection<KV<String, Integer>> counts =
   PCollectionList.of(mainInputCounts).and(incompleteCounts)
     .apply(Flatten.pCollections())
     .apply(Sum.integersPerKey());

counts.apply(ParDo.of(new DoFn() {
  @ProcessElement
  void processElement(ProcessContext c) {
    if (c.pane().isDrain()) {
       outputToIncompleteSink();
    } else {
      outputToCompleteSink();
   }}
  }
}));
```

(BTW if we added support to Partition to automatically divide a PCollection into drain and non-drain collections, we could simplify the above code).

State API

Beam also allows users to store durable, per-key state, and we need to figure out how this interacts with Drain.

In many scenarios, state is used to augment other aggregations, and therefore doesn't really need to be drained. For example, imagine a pipeline that calculates per-key, one-second histogram counts. This pipeline wants to detect portions of each histogram that are anomalous. It does so by adding a stateful DoFn that stores a model of what normal traffic for each key looks like. For example:

```
PCollection<KV<String, Integer>> mainInputCounts = p.apply(read())
   .apply(Window.into(FixedWindows.of(Duration.standardMinutes(10))))
```

```
    .apply(WithKeys((UserEvent e) -> e.getUserId()))
    .apply(Count.perKey());
PCollection<HistogramBucket> buckets = mainInputCounts
  .apply(ParDo.of(new DoFn() {
    private static final String STATE_TAG = "model";
    @ProcessElement
    void processElement(ProcessContext c,
                        BoundedWindow window,
                        @StateId(STATE_TAG) ValueState<Model> modelState) {
      // Read stored model for this key and update it.
      Model model = modelState.read();
      model.update(c.element());
      // Create a histogram bucket, augmented with anomalous information.
      HistogramBucket bucket = new HistogramBucket(
        c.element().getKey(),
        c.element().getValue(),
        window,
        model.isAnomalous(c.element().getValue(), window));
      c.output(bucket);
    }
  }));
buckets.apply(WriteOutput());
```

In this case, it's possible that losing the state on Drain is ok. This state is used simply to augment other aggregations, and it will be built again from scratch in a new pipeline. Of course it may still be unfortunate to lose these models. In the above case the models might need many days of data before they are accurate[6], so if they are thrown away the new pipeline might be stuck with poor models for a period of time.

Another common use case is using the state API to perform aggregations that are a bit awkward to do with the pure windowing model. Such a pipeline will store state and set timers to produce those aggregations downstream. Since all watermarks advance on Drain, these timers will all fire. Just as above in the PaneInfo section, the timer will need to know whether it is firing due to a Drain or not, and we could add this to OnTimerContext. However this might be an argument for simply adding this bit to WindowedContext, as that could be used by both processElement and onTimer; in this case we would not need to make it part of the PaneInfo.

We could also add an onDrain method that is always called when a pipeline is drained. The user could use this to save all state somewhere external. A more general solution would be to allow a general callback when a window is finished and state is about to be garbage collected (as discussed here). This is useful outside of drain, and can be used in drain as follows:

```
@OnWindowExpiration
Public void onExpiration(WindowedContext c) {
```

[6] In particular, since every day of the week is different, we might need at least 7 days worth of data before we have a good model.

```
  If (c.isDrain()) {
    // Save data outside of the pipeline.
  }
}
```

Source API

Unbounded sources advance their watermarks to infinity, and stop producing data. Effectively this operation turns unbounded sources into bounded sources.

Bounded sources should continue producing data until they are done. Often bounded sources in streaming pipelines are used as side inputs, and no data can be processed on the main input until the entire bounded source is read. As there is no obvious stopping point, we should just finish all bounded sources.

Splittable DoFn

Unbounded sources stop reading data, and regular DoFns drain out all data. What should splittable DoFn do? With splittable DoFn the distinction between sources and DoFns has been blurred, so what is the desired behavior?

A splittable DoFn always must know whether it's per-element output is bounded or unbounded (specified with the `@BoundedPerElement` and `@UnboundedPerElement` annotations). A splittable DoFn that is unbounded per element should be treated as an unbounded source by default - the runner advances watermarks and stops triggering elements. If however the DoFn is bounded per element, it should continue to run to completion.

The OnWindowExpiration callback can be used in a Splittable DoFn as well for custom processing when drain is triggered.

Snapshots

Pipeline snapshots and Update will be a separate Beam proposal, but since they are particularly useful during Drain we will discuss them briefly here.

A snapshot is a point-in-time snapshot of a pipeline's state that can be restored to the same (or a modified) pipeline at some time in the future. The proposal is to also allow partial restore of a snapshot - i.e. start a new pipeline, but only restore specific pipeline steps from the snapshot.

This is particularly useful with regards to custom sources and splittable DoFns. Imagine a custom source reading from Kafka. The current location in all the Kafka partitions is stored persistently in the Beam sources. After draining the pipeline a user will want to start a new

pipeline where the last one “left off,” however there’s no good way of figuring out what those positions are. Today the only way of doing this is for the source to constantly track positions external to the pipeline (done automatically with Google Cloud Pub/Sub, and in Zookeeper for Kafka).

However if the user snapshots the pipeline before draining it[7], the source state is available in the snapshot. If the user then starts a new pipeline with the same sources, they can restore just the source transforms from the snapshot to start the new pipeline from where the last pipeline left off.

Failing Elements

Drain is not intended to handle persistently failing elements (i.e. user code throws an unhandled exception each time the element is seen). In this case, the pipeline will get stuck attempting to drain the failing element. The best way to address this use case is to fix the user code and update the pipeline. A subsequent Beam proposal will talk about ways to in-place update pipeline code to address these types of situations.

[7] The user will want to pause the pipeline first.