Implemented in [#2455](#2455) *TL;DR Get rid of DoFn.ProcessContinuation, stop()/resume() methods, report output watermark via ProcessContext continuously along with the output itself.*
Partially reverted in [#3360](#3360) *after figuring out a sensible semantics for resume()*

Output watermark reporting

Currently an SDF reports a watermark on its future output by returning a `ProcessContinuation`, e.g.: `return resume().withFutureOutputWatermark(...)`.

This is problematic for several reasons:

- This assumes that the watermark is needed only after the `@ProcessElement` call completes - and that the call *completes at all*. But a runner like Flink could let a `@ProcessElement` call run for a very long time, and would likely still be interested in knowing what timestamps of future outputs from this currently running call will be - i.e. in **continuously updating the watermark while the call runs**.
  I.e., the current API restrictively assumes a micro-batch implementation.

- If a runner performs a dynamic split on the running `@ProcessElement` call, there is simply no place in the API to report a watermark for the residual restriction.
  The current API is unnecessarily focused on checkpointing - i.e. it assumes that the residual restriction becomes relevant only once the `@ProcessElement` call completes, but this is not always the case: **residual restriction, and its watermark, become relevant immediately when a split happens**.

- The SDF might be outputting into multiple `PCollections`, and it might want to make different promises about outputting to each of them - i.e. **watermark reporting should be per output tag**.
- The entire ProcessContinuation API is ill-defined, for reasons I outline below.

Because of this, I suggest to report the watermark differently:

- Add methods `ProcessContext.updateWatermark(Instant)` and `updateWatermark(OutputTag<?>, Instant)`

- Runners will use this to continually have access to the most up-to-date promise from the `DoFn` about timestamps of future output from the current call.
  In practice, current runners tend to be interested only in the most recent value when the `@ProcessElement` call completes, and they commit it together with the output from the call, but in principle, this API will allow runners to potentially do more.

- This applies not only to SDF, but also e.g. to DoFn's that use timers to resume the call later.

- There are some cases (e.g. Google Cloud PubSub) where it'd be useful to allow calling these methods concurrently to `c.output()`, e.g. by polling the third-party service for a watermark asynchronously in a thread.

With this, e.g. the relevant part of KafkaIO on SDF will look something like this:

```
@ProcessElement
public void process(ProcessContext c, OffsetRangeTracker tracker) {
      KafkaClient client = Kafka.connect(c.element());
      client.seek(tracker.start());
      while (true) {
            KafkaMessage message = client.poll();
            if (!tracker.tryClaim(message.offset())) return;
            c.updateWatermark(message.timestamp());
            c.output(message);
      }
}
```

ProcessContinuation, and returning from @ProcessElement

While implementing SDFs for testing purposes, I often found myself wondering when I should return `stop()` vs. `resume()` from the `@ProcessElement` call.

First, for reasons above, "future output watermark" should not be part of ProcessContinuation. All that remains is the `stop()` / `resume()` signal.

I think the proper way to treat them is as follows:

- `stop()` means "I have completed all the work associated with the current value of `tracker.currentRestriction()`".
  ⇒ after `tracker.tryClaim()` returned false, you have to return `stop()`.

- `resume()` means "I have **not** completed all the work associated with the current value of tracker.currentRestriction()".
  Then, runner needs to know how much work you *have* completed and how much yet needs to be done - that's what `RestrictionTracker.checkpoint()` is for.
  This is also the reason why a `checkpoint()` method is necessary, in addition to a `splitAfterFractionOfRemainder()` method.
- Generally a `@ProcessElement` method of an SDF is a loop, where on each iteration you:
  - `tryClaim()` a block - process it if successful, or return `stop()` otherwise
  - or you voluntarily return `stop()` because you somehow know that there's nothing more to `tryClaim()` within this restriction

   - or you voluntarily return `resume()` because you somehow know that there's currently nothing more to `tryClaim()` within this restriction, but there might be later

However, having `resume()` puts us into a tricky situation implementation-wise. Suppose that `@ProcessElement` was running, and runner already took a checkpoint because the call emitted too much output, or ran for too long. So the runner already has a residual restriction.

Now, suppose that the call voluntarily returns `resume()`! Now the runner has **two residual restrictions** - one for the work it already split off, one for the work that the call just said it didn't complete; both need to be processed.

Current runner-agnostic implementation of SDF, and most current runners, can't really do this easily - the current implementation uses a sequential checkpoint loop - i.e. when `@ProcessElement` completes, it assumes that there's exactly 1 residual restriction (checkpoint to resume from), and sets a timer to process it.

I would like to emphasize that **the current implementation of SDF is buggy in this respect** - the only reason we didn't notice is that current SDF unit tests are themselves buggy: they don't return `stop()` after a failed `tryClaim`.

Moreover, if `resume()` specified a resume delay, it's not clear which of these two residual restrictions it refers to. Also, in some cases this “two residual restrictions” situation can lead to an exponential proliferation of residual restrictions.

Because of all this, **let’s remove ProcessContinuation** and say that SDF.`@ProcessElement`:

- must return `void`
- must guarantee that once it returns, the entire `tracker.currentRestriction()` has been processed (even though the restriction of course might have changed while the call ran).

For use cases where one wants to resume processing later - I propose to (at least temporarily) declare that out of scope of SDF, and let people just use the timers API.

Overall

- Add `c.updateWatermark()` and report watermark continuously via this method.
- Make SDF.`@ProcessElement` return void, which is simpler for users though it doesn't allow to resume after a specified time
- Declare that SDF.`@ProcessElement` must guarantee that after it returns, the entire `tracker.currentRestriction()` was processed.
- Add a `bool RestrictionTracker.done()` method to enforce the bullet above.
- For resuming after specified time, use regular `DoFn` with state and timers API.