BEAM SUMMIT

Complex Event Processing with State and Timers

Israel Herraiz, Miren Esnaola
Google

BEAM SUMMIT

NYC 2023

Hands-on

- Have a Python development environment ready, with Python 3.10
- Use your favourite IDE (Visual Studio Code, PyCharm, …)
- Clone the workshop repo

  github.com/iht/beam-state-timers-quickstart

- Install dependencies with

```
pip install -r requirements.txt
```

- Run the tests (they fail until you write the solution 😄) with `pytest`.
- Have a look at the file `my_pipeline/pipeline.py` and fill the gaps

Agenda

- What is stateful stream processing?
- Uses cases
- State properties
- Types of state
- Example
- Timers

Stateful stream processing — What is it?

Stateful stream processing is a subset of stream processing in which the computation maintains contextual state. This state is used to store information derived from the previously-seen events.

Stateful stream processing — Use cases

Most non-trivial stream processing applications require stateful event processing:

Personalization

A video streaming service could use it to track a user’s past viewing history and use this information to make recommendations for movies and TV shows.

Fraud detection

A financial institution could use it to track a user’s past transactions and use this information to detect unusual activity that might indicate fraud.

Supply chain management

A logistics company could use it to track the location and status of packages in real-time, and to optimize the routing of packages based on past delivery times and delays.

State Properties

It is identified by the name that must be **unique** through the transform.

It must remain **local** to the transform.

It can contain different types of objects: scalar values, collections or maps.

It works **per key**.

It is **bound to a window**.

How it is stored **depends on the runner** implementation.
For the Direct Runner state is stored in memory.

Types of State

ReadModifyWriteState

A readable state cell containing a single value.

CombiningValueState

A readable state cell defined by a function, accepting multiple input values, combining then and producing a single output value.

BagState

A readable state cell containing a bag of values. Items can be added to the bag and the contents read out.

Types of State — ReadModifyWriteState

```python
class ReadModifyWriteStateDoFn(DoFn):

  STATE_SPEC = ReadModifyWriteStateSpec('num_elements', VarIntCoder())

  def process(self, element, state=DoFn.StateParam(STATE_SPEC)):
    # Read the number element seen so far for this user key.
    current_value = state.read() or 0
    state.write(current_value + 1)

_ = (p | 'Read per user' >> ReadPerUser()
       | 'state pardo' >> beam.ParDo(ReadModifyWriteStateDoFn()))
```

Types of State — CombineValueState

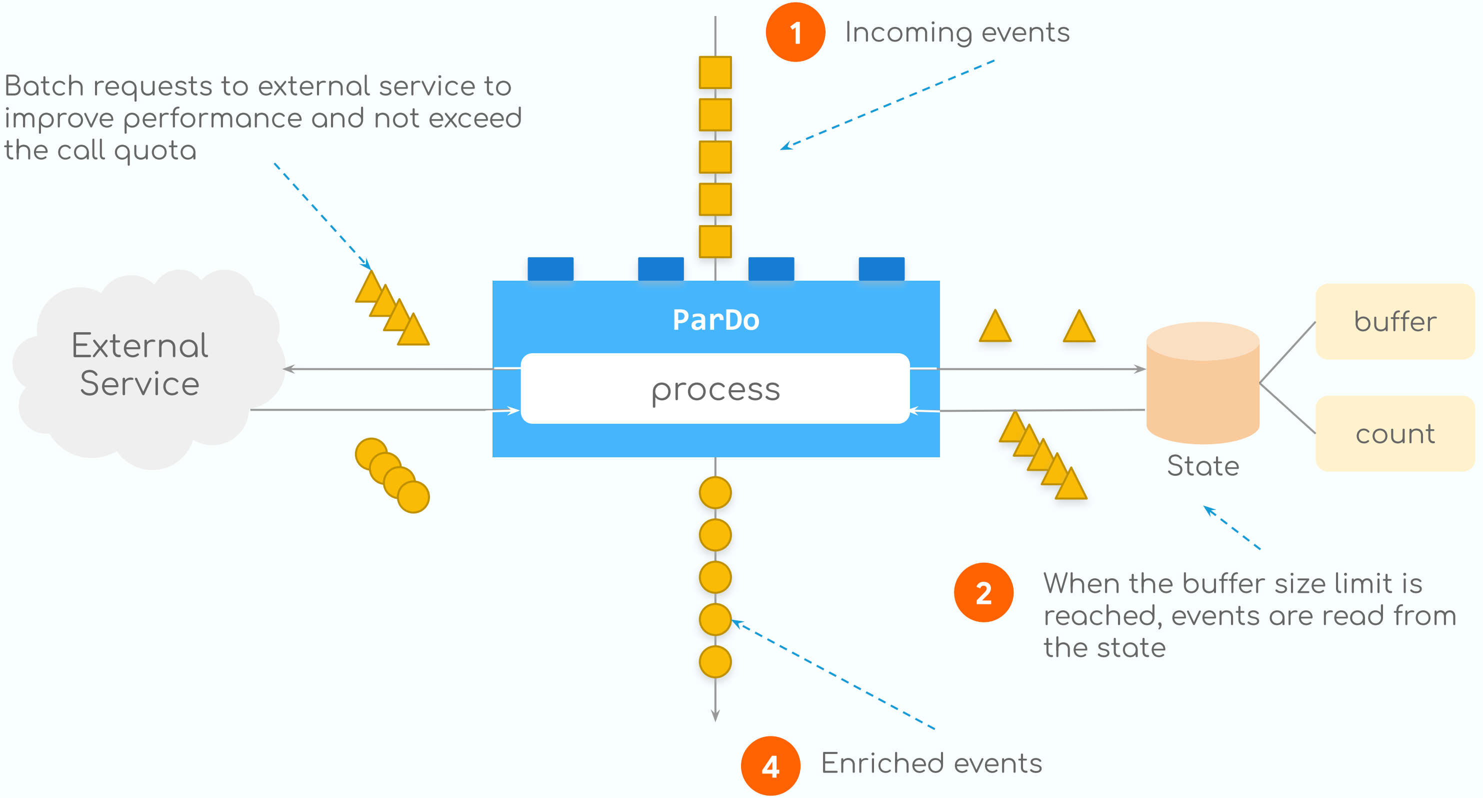
```python
class CombiningStateDoFn(DoFn):
  SUM_TOTAL = CombiningValueStateSpec('total', sum)
  def process(self, element, state=DoFn.StateParam(SUM_TOTAL)):
    state.add(1)

_ = (p | 'Read per user' >> ReadPerUser()
       | 'Combine state pardo' >>
beam.ParDo(CombiningStateDoFn()))
```

Types of State — BagState

```python
class BagStateDoFn(DoFn):
  ALL_ELEMENTS = BagStateSpec('buffer', coders.VarIntCoder())

  def process(self, element_pair, state=DoFn.StateParam(ALL_ELEMENTS)):
    state.add(element_pair[1])
    if should_fetch():
      all_elements = list(state.read())
      process_values(all_elements)
      state.clear()

_ = (p | 'Read per user' >> ReadPerUser()
       | 'Bag state pardo' >> beam.ParDo(BagStateDoFn()))
```

Umm, let’s look at a stateful event processing *example…*
Suppose you have a pipeline with a ton of data coming in and you need to enrich each event by with the information returned by a external system. You can’t just issue a call per event, performance would be terrible and it would blow your quota with that external system. What shall you do?

Example



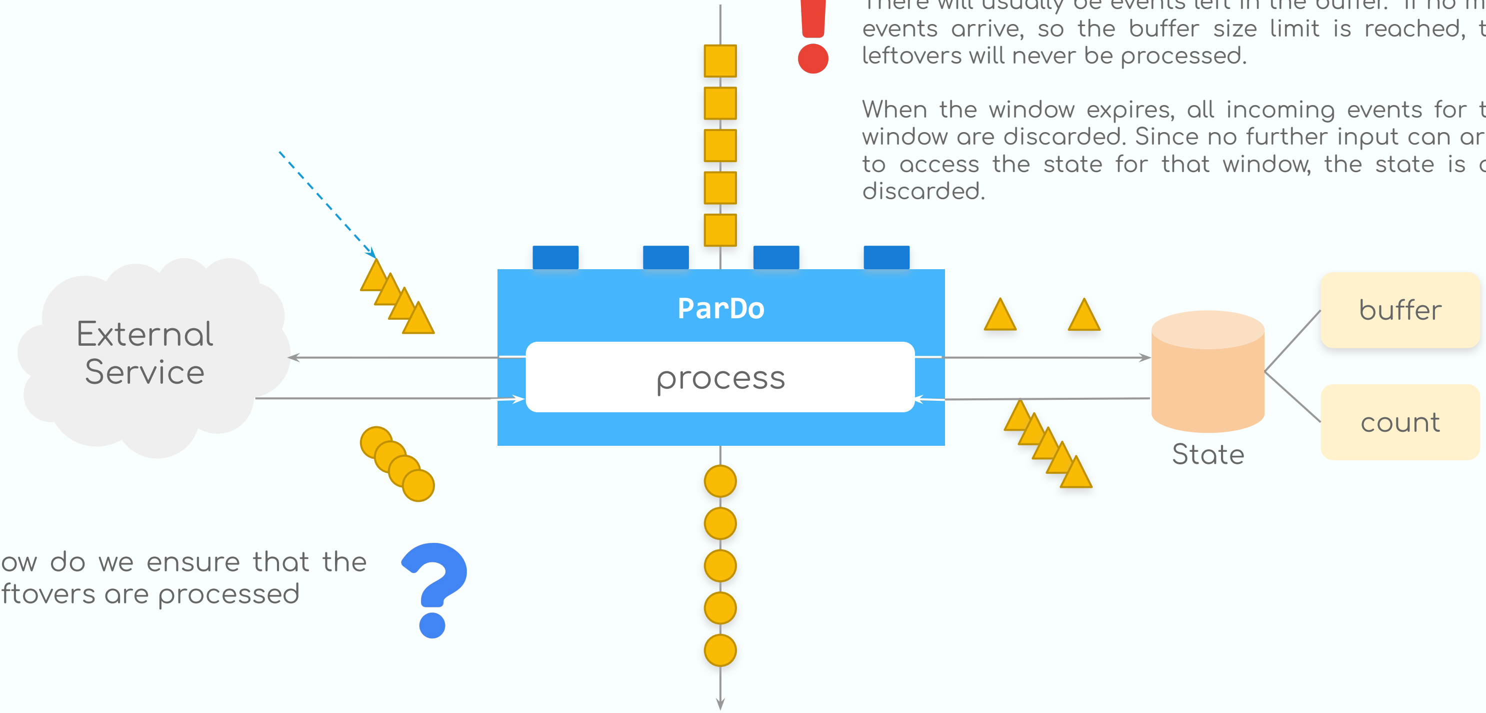
1 Incoming events

2 When the buffer size limit is reached, events are read from the state

3 Batch requests to external service to improve performance and not exceed the call quota

4 Enriched events

Example



There will usually be events left in the buffer. If no more events arrive, so the buffer size limit is reached, that leftovers will never be processed.

When the window expires, all incoming events for that window are discarded. Since no further input can arrive to access the state for that window, the state is also discarded.

How do we ensure that the leftovers are processed ?

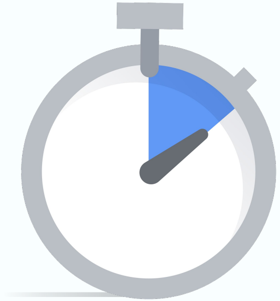
Example

```
class StatefulBufferingFn(beam.DoFn):
  MAX_BUFFER_SIZE = 500;
  BUFFER_STATE = BagStateSpec('buffer', EventCoder())
  COUNT_STATE = CombiningValueStateSpec('count',
                                        VarIntCoder(),
                                        combiners.SumCombineFn())

  def process(self, element,
              buffer_state=beam.DoFn.StateParam(BUFFER_STATE),
              count_state=beam.DoFn.StateParam(COUNT_STATE)):
    buffer_state.add(element)
    count_state.add(1)
    count = count_state.read()
    if count >= MAX_BUFFER_SIZE:
      for event in buffer_state.read():
        yield event
      count_state.clear()
      buffer_state.clear()
```

Timers

Event-time Timers

Callback when the watermark reaches some threshold.



Processing-time Timers

Callback after a certain amount of time has elapsed.

Timers — Event-time Timer

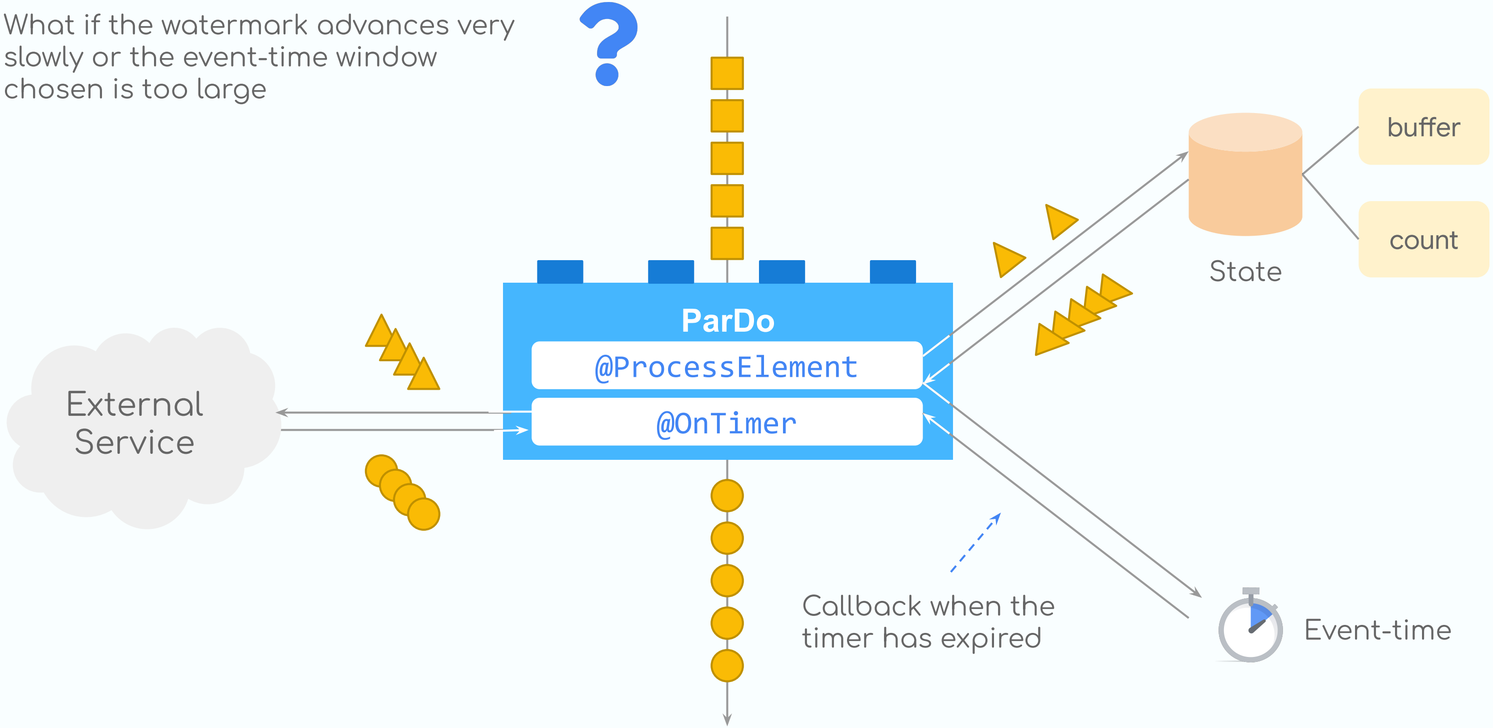
```
class EventTimerDoFn(DoFn):

  ALL_ELEMENTS = BagStateSpec('buffer', coders.VarIntCoder())
  TIMER = TimerSpec('timer', TimeDomain.WATERMARK)

  def process(self,
              element_pair,
              t = DoFn.TimestampParam,
              buffer = DoFn.StateParam(ALL_ELEMENTS),
              timer = DoFn.TimerParam(TIMER)):
    buffer.add(element_pair[1])
    # Set an event-time timer to the element timestamp.
    timer.set(t)

  @on_timer(TIMER)
  def expiry_callback(self, buffer = DoFn.StateParam(ALL_ELEMENTS)):
    state.clear()


_ = (p | 'Read per user' >> ReadPerUser()
       | 'EventTime timer pardo' >> beam.ParDo(EventTimerDoFn()))
```

Timers — Processing Timer

```
class ProcessingTimerDoFn(DoFn):

  ALL_ELEMENTS = BagStateSpec('buffer', coders.VarIntCoder())
  TIMER = TimerSpec('timer', TimeDomain.REAL_TIME)

  def process(self,
              element_pair,
              t = DoFn.TimestampParam,
              buffer = DoFn.StateParam(ALL_ELEMENTS),
              timer = DoFn.TimerParam(TIMER)):
    buffer.add(element_pair[1])
    # Set a timer to go off 30 seconds in the future.
    timer.set(Timestamp.now() + Duration(seconds=30))

  @on_timer(TIMER)
  def expiry_callback(self, buffer = DoFn.StateParam(ALL_ELEMENTS)):
    state.clear()


_ = (p | 'Read per user' >> ReadPerUser()
       | 'EventTime timer pardo' >> beam.ParDo(ProcessingTimerDoFn()))
```

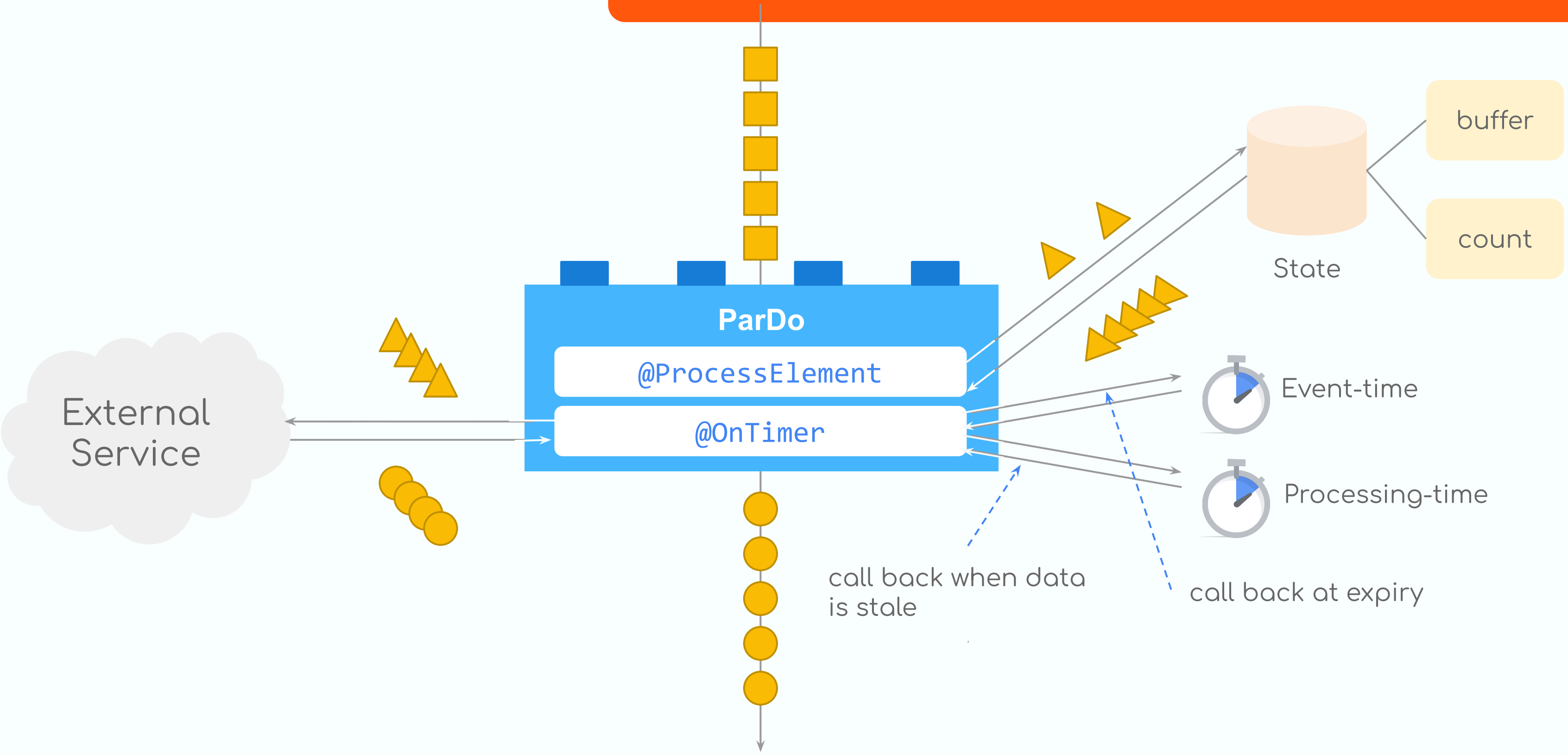
Timers

What if the watermark advances very slowly or the event-time window chosen is too large



Callback when the timer has expired

Timers


buffer
count
State
ParDo
@ProcessElement
@OnTimer
External Service
Event-time
Processing-time
call back when data is stale
call back at expiry

Timers

```
class StatefulBufferingFn(beam.DoFn):
  ...
  STALE_TIMER = TimerSpec(stale, TimeDomain.REAL_TIME)

  def process(self, element,
              w=beam.DoFn.WindowParam,
              buffer_state=beam.DoFn.StateParam(BUFFER_STATE),
              count_state=beam.DoFn.StateParam(COUNT_STATE),
              expiry_timer=beam.DoFn.TimerParam(EXPIRY_TIMER),
             stale_timer=beam.DoFn.TimerParam(STALE_TIMER)):
    if count_state.read() == 0:
    # We set an absolute timestamp here (not an offset like in the Java SDK)
      stale_timer.set(time.time() + StatefulBufferingFn.MAX_BUFFER_DURATION)
    ... same logic as above ...

  @on_timer(STALE_TIMER)
  def stale(self,
             buffer_state=beam.DoFn.StateParam(BUFFER_STATE),
             count_state=beam.DoFn.StateParam(COUNT_STATE)):
    events = buffer_state.read()

    for event in events:
      yield event

    buffer_state.clear()
    count_state.clear()
```

added an additional processing-time timer in case the buffer is filling too slowly

It’s time for a lab, folks!!!

New York Taxi Vectors by Vecteezy

Input messages

```
{
    "latitude" : 40.77405,
    "longitude" : -73.9638,
    "meter_increment" : 0.024726477,
    "meter_reading" : 6.428884,
    "passenger_count" : 5,
    "point_idx" : 260,
    "ride_id" : "ccf021d0-ec37-41f1-9637-cf8bcfbcbb2d",
    "ride_status" : "enroute",
    "timestamp" : "2023-06-11T09:40:03.15611-04:00"
}
```

Output messages

```
{
    "ride_id" : "ccf021d0-ec37-41f1-9637-cf8bcfbcbb2d",
    "start_time" : "2023-06-11T09:40:03.15611-04:00"
    "end_time" : "2023-06-11T15:47:03.16611-04:00"
    "start_status" : "pickup"
    "end_status" : "dropoff"
    "ride_duration_in_secs" : 367
    "reason" : "DROPOFF_SEEN"
    "n_points" : 12
}
```

Hands-on

- Have a Python development environment ready, with **Python 3.10**
- Use your favourite IDE (Visual Studio Code, PyCharm, ...)
- Clone the workshop repo

  github.com/iht/beam-state-timers-quickstart

- Install dependencies with

```
pip install -r requirements.txt
```

- Run the tests (they fail until you write the solution 😄) with `pytest`.
- Have a look at the file `my_pipeline/pipeline.py` and fill the gaps

QUESTIONS?

Israel Herraiz
linkedin.com/in/herraiz
github.com/iht

Miren Esnaola
linkedin.com/in/mirenesnaola
github.com/apichick