Buffering Data by Timestamp: A Step Towards Time Series Processing

Shunping Huang & Claude van der Merwe

Tuesday, June 23, 2026

What we'll cover

1. **Introduction**
   Process an event only after the earlier ones, using them as context

2. **Beam primitives** — fast primer
   Event time, watermarks, state & timers

3. **The simple approach**
   Windows + combiners, and limitations

4. **The stateful approach** — the deep dive
   timers, watermarks, batching

5. **Benchmarks**

Per-event processing on an ordered stream

For each event, compute something over its previous N events, in event-time order, for example an anomaly score or a forecast from a model like TimesFM.

Forecasting

Predict the next value from the recent window, as models like TimesFM do.

context: recent values

Anomaly detection

Flag a reading that breaks from the device's recent baseline.

context: recent readings

Fraud scoring

Judge a transaction against the account's recent activity.

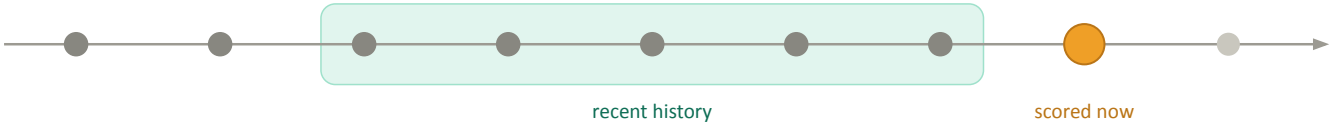
context: recent transactions

The same shape across domains: each event processed in event-time order, with its recent history as context. The challenge is doing it efficiently at stream scale.

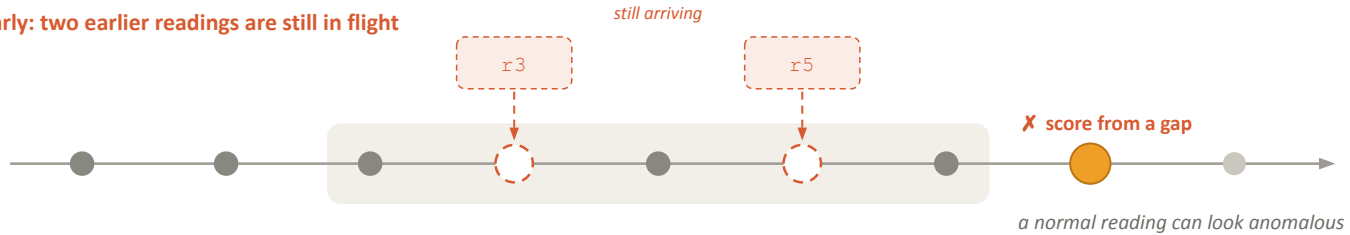
Processing needs complete history

To score an event, you look at the readings just before it. If they have not all arrived yet, the score is computed against a gap.

What we want: each event scored against the readings just before it



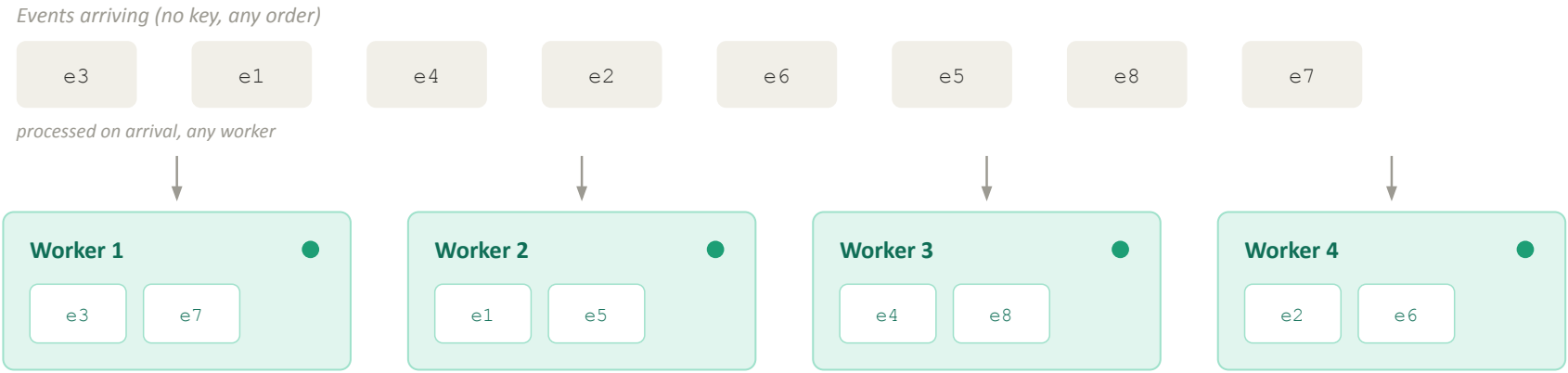
Process too early: two earlier readings are still in flight



To process an event correctly, the data must be complete up to its time. Process too early and you compute against partial history.

Unkeyed and unordered: fully parallel

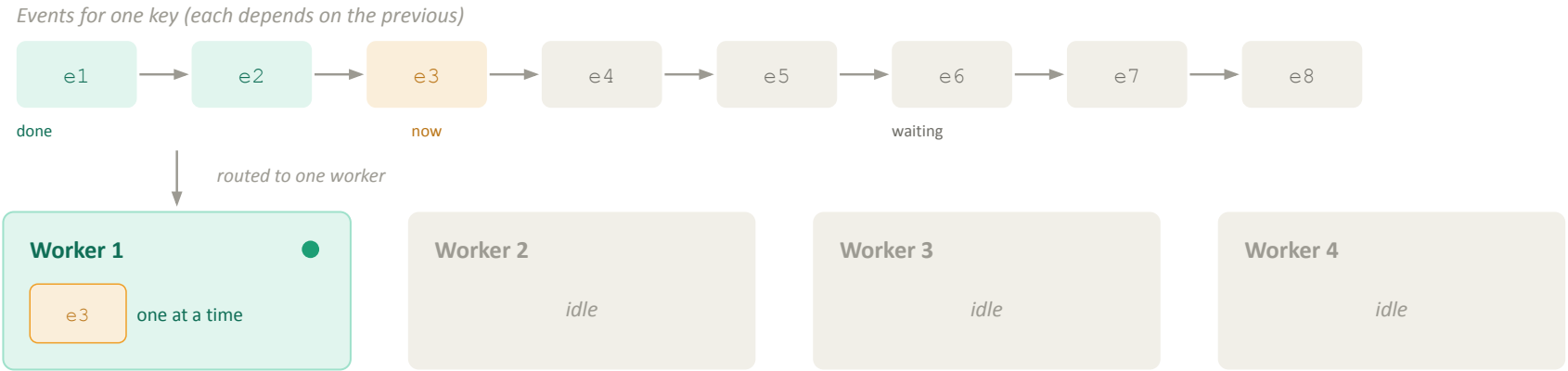
Process any element the instant it arrives, on any worker. Nothing depends on anything, so nothing waits.



Process on arrival, anywhere. Nothing blocks, so throughput scales with the number of workers.

Keyed and ordered: must wait to process

Now an element depends on the ones before it, so you cannot process on arrival. Each one waits until the data is complete up to its time.



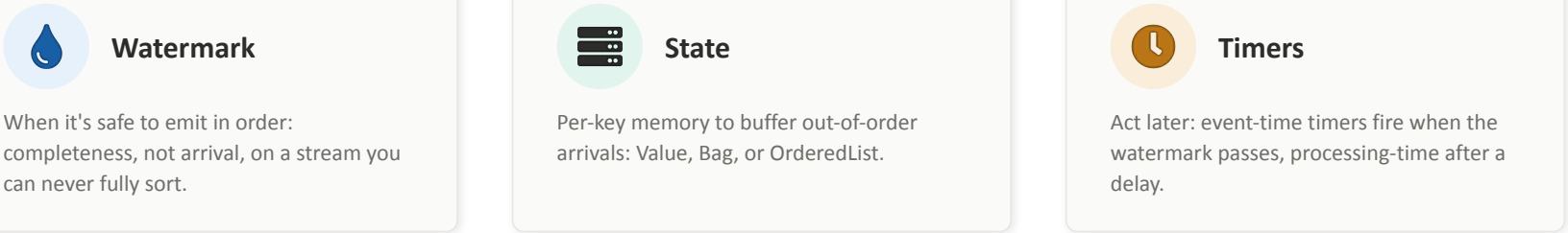
Can't process on arrival: each element waits until the data is complete up to its time. That wait serializes the key onto one worker, the others sit idle. Single-key throughput is the ceiling.

Parallel across keys, serial within one

We just saw why: ordering serializes a key onto a single worker.

Across keys, Beam stays parallel, so you scale out.
Within a key, single-key throughput is the ceiling. Beam gives you the tools to do the serial part correctly:

The tools Beam already gives you:

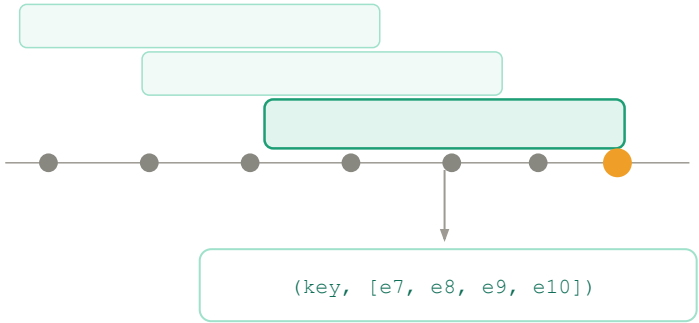


Sliding windows with a combiner

Windows group a stream into event-time buckets that fire when the watermark passes. Size a sliding one to your lookback, slide it one event at a time.

```
(events
 | beam.WindowInto(
     SlidingWindows(size, period))
 | beam.CombinePerKey(
     ToListCombineFn())
 | beam.Map(lambda k, vs: (k,
     sorted(vs, key=event_time))))
```

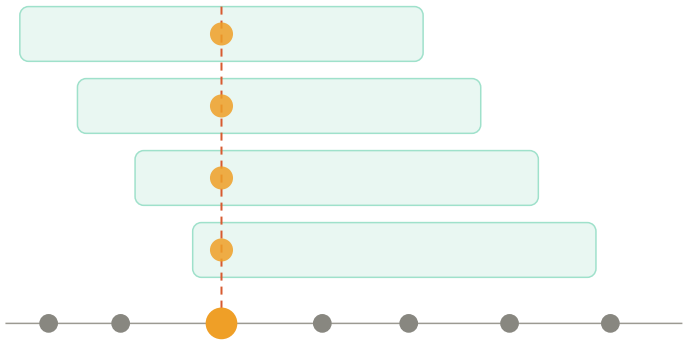
each window → event + previous K, sorted



- ✔ No state, no timers
- ✔ Watermark gating + cleanup, free

What the windowing approach costs

One event lands in K overlapping windows



= ×K copies on the wire

Duplication. Each event is copied into ~K overlapping windows.

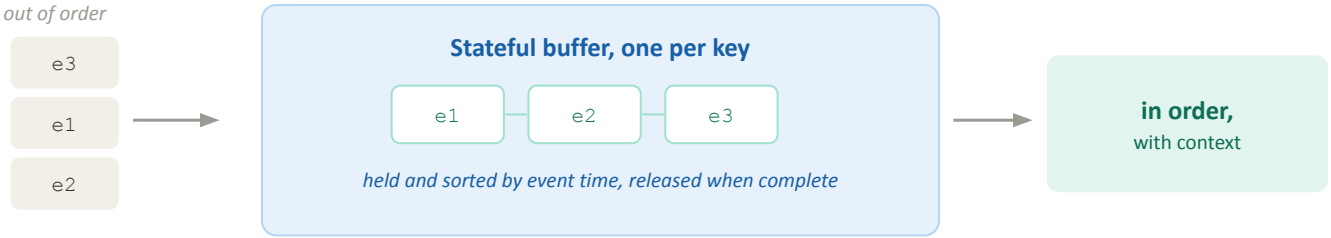
Shuffle blowup. All those copies churn through GroupByKey, the bytes balloon.

No control of the frame. The window decides what your computation sees.

Windowing gives you no control over the per-event context, and copies each event across many windows. **A stateful buffer avoids both.** →

A reusable stateful buffer DoFn

Route each key's events to one stateful DoFn. It buffers them per key and releases them in event-time order, each with its recent history as context.



Each call hands your code the event, **its key, timestamp, and value, plus its context: the previous N events, already in order.**

Build it once, reuse it for any key. State backends: BagState on any runner, OrderedListState on Dataflow for faster range reads.

Ordering with a looping event-time timer

You can't read the watermark inside a DoFn, but a firing event-time timer tells you it just passed.

1 Arm

At the earliest unprocessed element, set an event-time timer at its timestamp.

2 Fire

The watermark reaches that timestamp: the data is complete up to it, so the timer fires.

3 Process

The element is now safe. Process it with its context and mark it done.

4 Advance

Move to the next earliest unprocessed element, and arm a timer at it.

repeat: advance to the next earliest unprocessed, one element per fire

One turn of the loop:

e0, e1 done.

Timer armed on e2, the earliest unprocessed. When the watermark reaches it, it fires and e2 is processed.

The per-element loop doesn't scale

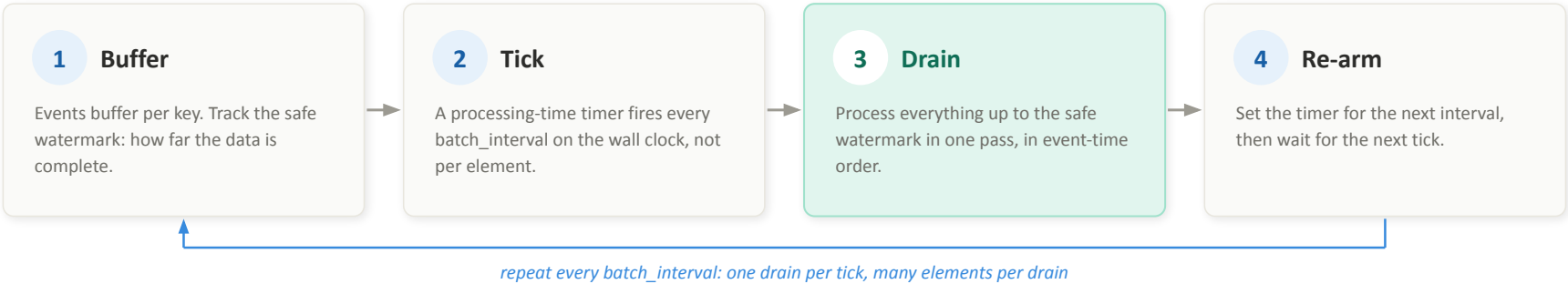
The loop is correct, but it does one full state round-trip per element.

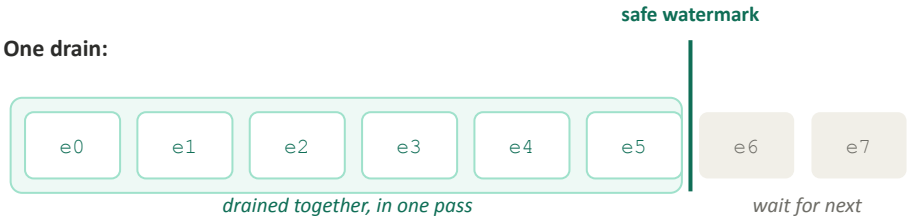
each element pays a 900ms+ overhead cost:

Diagnosis: one fire processes one element with one state round-trip; batching amortizes that cost.

Draining the buffer in batches

Stop processing one element per fire. Track how far it is safe to process, then drain that whole range on a timer.





Two clocks.

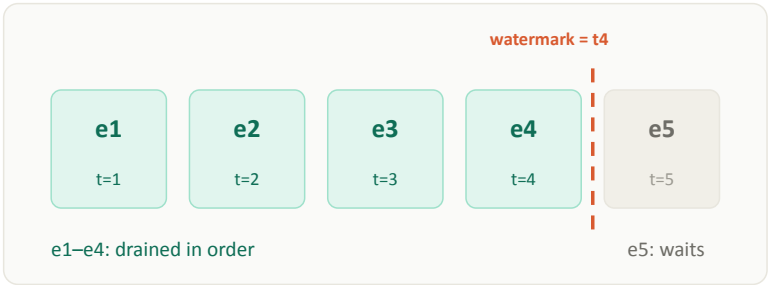
The watermark (event time) decides what is eligible; the batch timer (wall clock) decides when you drain. Many elements per drain, not one per fire.

A trace of one batched cycle

Events arrive out of order; batch_interval = 5s. One cycle, end to end.

1. **e3 (t=3) arrives**, earliest unprocessed, so arm the event time timer at t=3.
2. **e1, e5, e2, e4 follow, out of order**, all buffered; earliest is now e1 (t=1), timer re-points.
3. **Event time timer fires**, watermark reached the lower bound, so arm processing time BATCH_TIMER for now + 5s (wall clock).
4. **BATCH_TIMER fires**, process up to the watermark (t=4), in order: e1 e2 e3 e4; e5 stays.
5. **e5 still unprocessed**, arm the timer at t=5 ⟶ next cycle.

buffer: kept sorted by event time

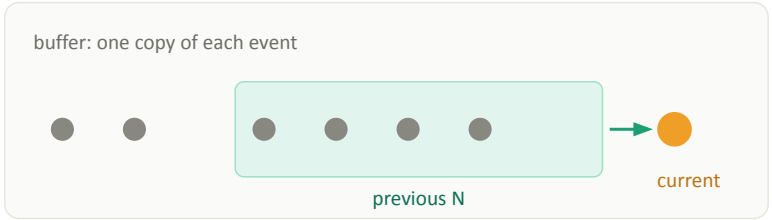


BATCH_TIMER fires +5s ⟶ drain everything ≤ t=4

Footnote on the probes: step 4 needs the current watermark, but Beam doesn't expose it to a DoFn. We estimate it with a few event-time timers (MIN/MID/MAX), a workaround for an API gap, not the idea. Ideally: read it directly and drain up to it.

Both requirements met: order and context

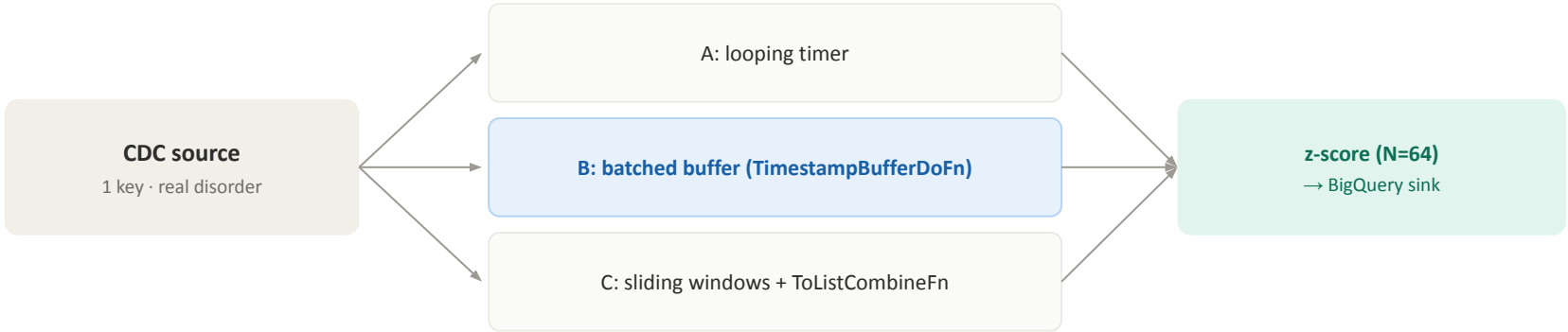
Order via the watermark, context via the buffer, each event stored once.



stored once, not ×K

How the benchmark is set up

Same input, same z-score, real out-of-order CDC, only the ordering mechanism differs.



- Single worker on Dataflow
- Context is 64 prior events
- Batches drain every 5s (approach B)
- 90k rows
- 15 min
- 100 rows per second

Throughput at 100 rows/s

Single key, 100 rows/s. Only the batched buffer keeps up, the other two are already underwater.

	Sustained	Keeps up?	Data freshness
A: looping timer	~1 / s	✗	243 → 776 s, growing
B: batched buffer	**100 / s**	✓	45 s, bounded
C: sliding windows	~40 / s	✗	410 → 597 s, growing

Only the batched buffer keeps up at 100/s, A falls ~100× short, C ~2.5×.

Latency includes a ~30–45 s CDC poll + buffer + batch floor, infrastructure, not the algorithm. One rate tested: B's true ceiling is above 100/s (untested); A's ~900 ms/element floor caps it above ~1/s.

Why each implementation hits its limit

Same z-score, same data, three different cost models.

A: looping timer

pays per element

~900 ms/element floor (timer + state + commit), rising to ~4 s as the backlog grows. Drained < 1% of input.

1 sort / element

B: batched buffer

pays per batch

One full-buffer sort per drain; one commit amortized across ~760 elements.

118 sorts / 90k

C: sliding windows

pays per copy

Each element fans into 64 overlapping windows.

5.76M copies

State operations and sort: A pays per element, C pays per copy, B pays per batch.

Takeaways

1

Ordered processing is a composable pattern built from Beam's primitives.

Buffer per key, gate on the watermark, drain in batches: one reusable stateful DoFn gives you order and context, exact across retries. You can build this today.

2

Batching the drain amortizes the per-element cost.

Per-element ordering is correct but pays a full state round-trip every element; draining a safe range on a timer makes it one pass per batch. At 100 rows/s, that was the difference between keeping up and falling ~100× behind.