The Dataflow Model

Problem

- How can we process unbounded data?
- Example: track user activity on a website

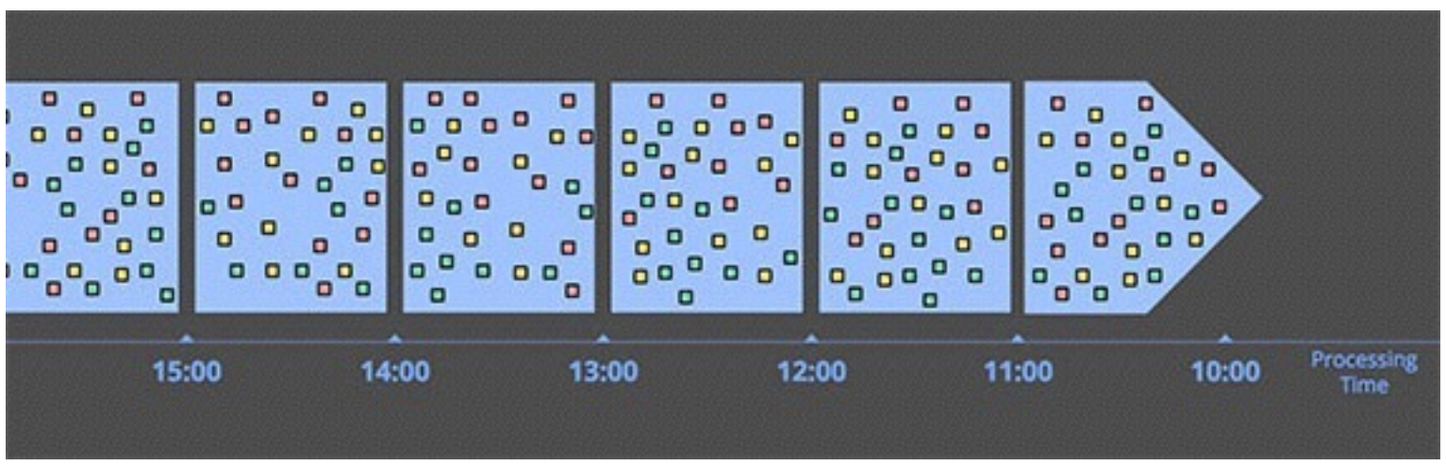
Key ideas

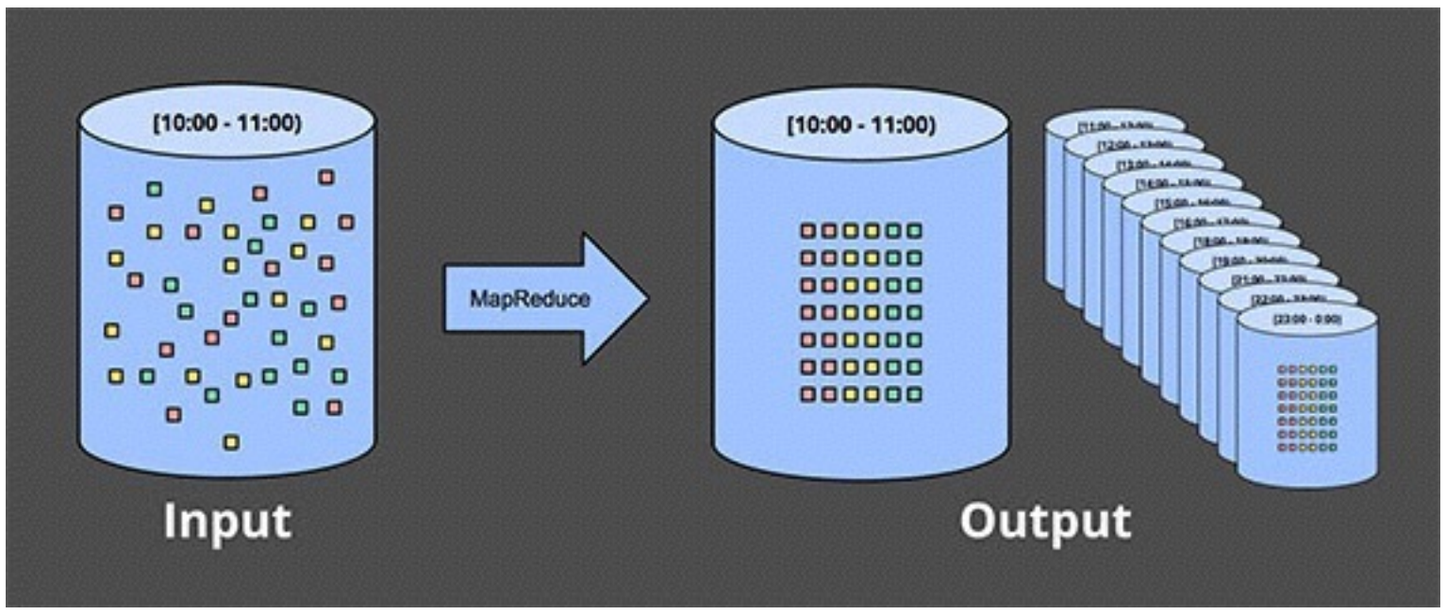
- **Windowing**
  - Fixed windows
  - Sliding windows
  - Sessions
- **Time domains**
  - Event time
  - Processing time
- **Triggers**

Contribution

- Dataflow API
  - Easily build pipelines with your choice of windowing, time domain, and trigger
- Independent of execution engine
  - Choose batch, micro-batch, or streaming depending on tradeoffs

Windowing

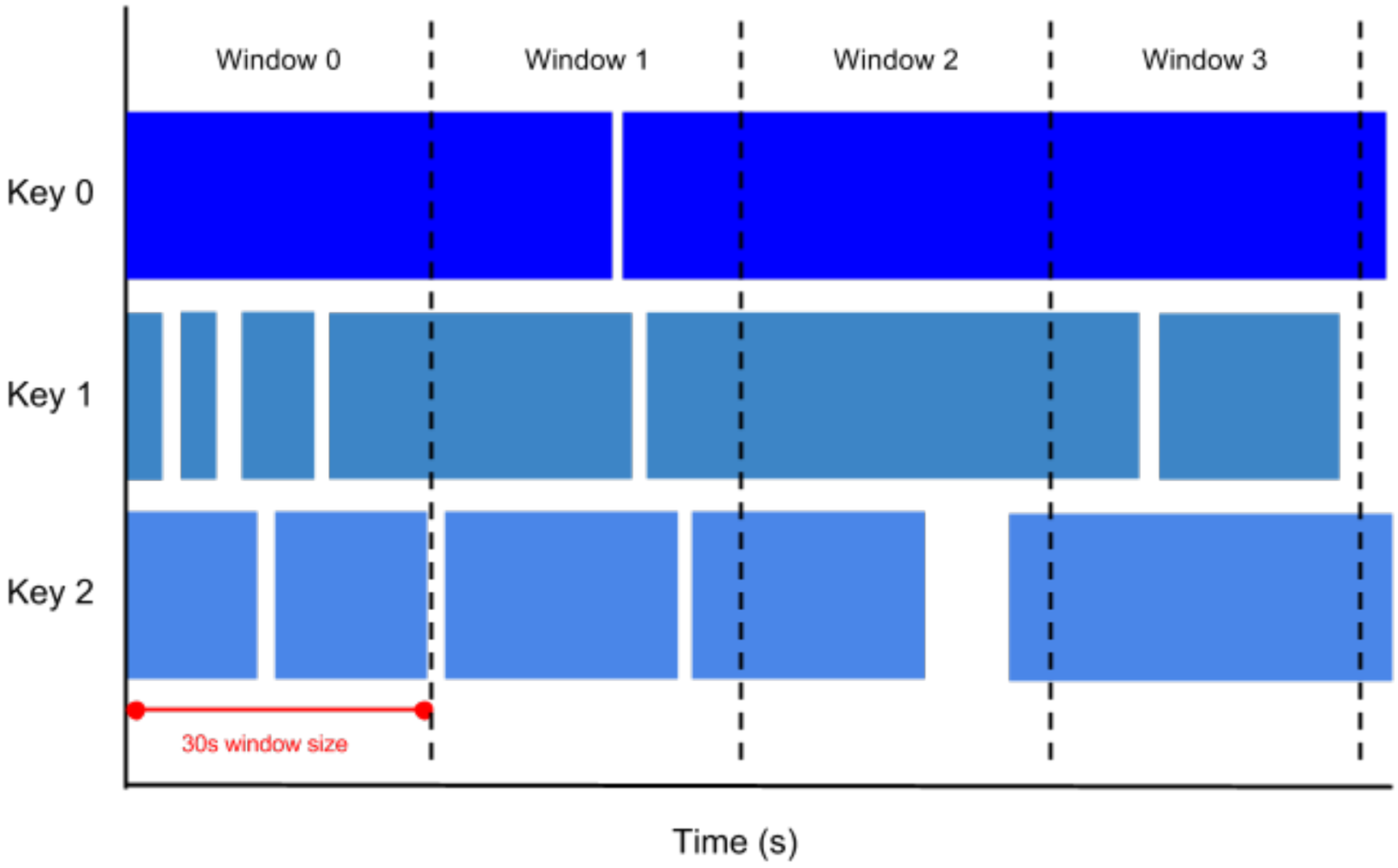

15:00
14:00
13:00
12:00
11:00
10:00
Processing Time


[10:00 - 11:00)
MapReduce
[10:00 - 11:00)
Input
Output

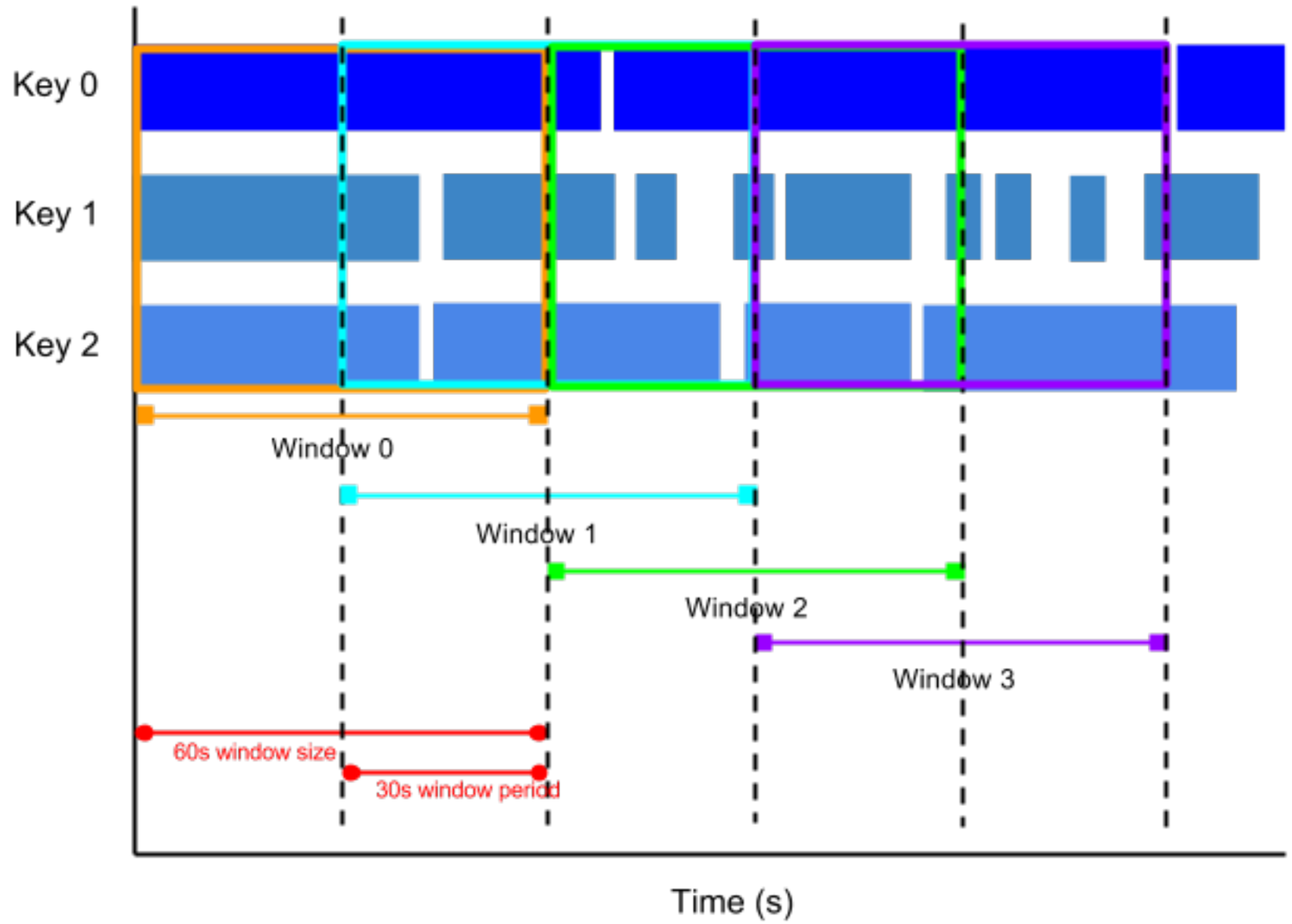
Types of windows

- Fixed windows
- Sliding windows
- Sessions

Fixed windows

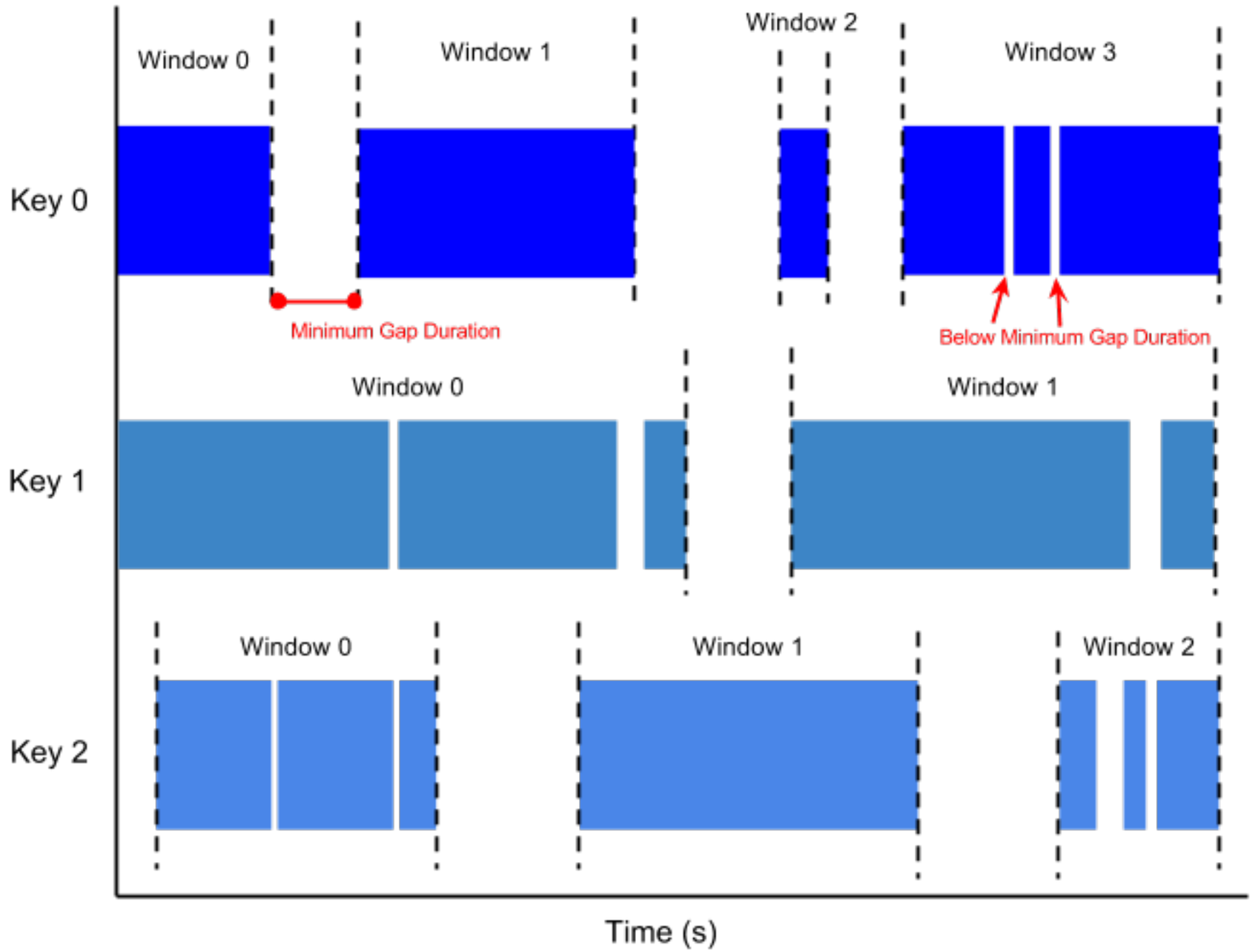


Sliding windows



Example: compute running average over past 5 minutes of data

Session windows

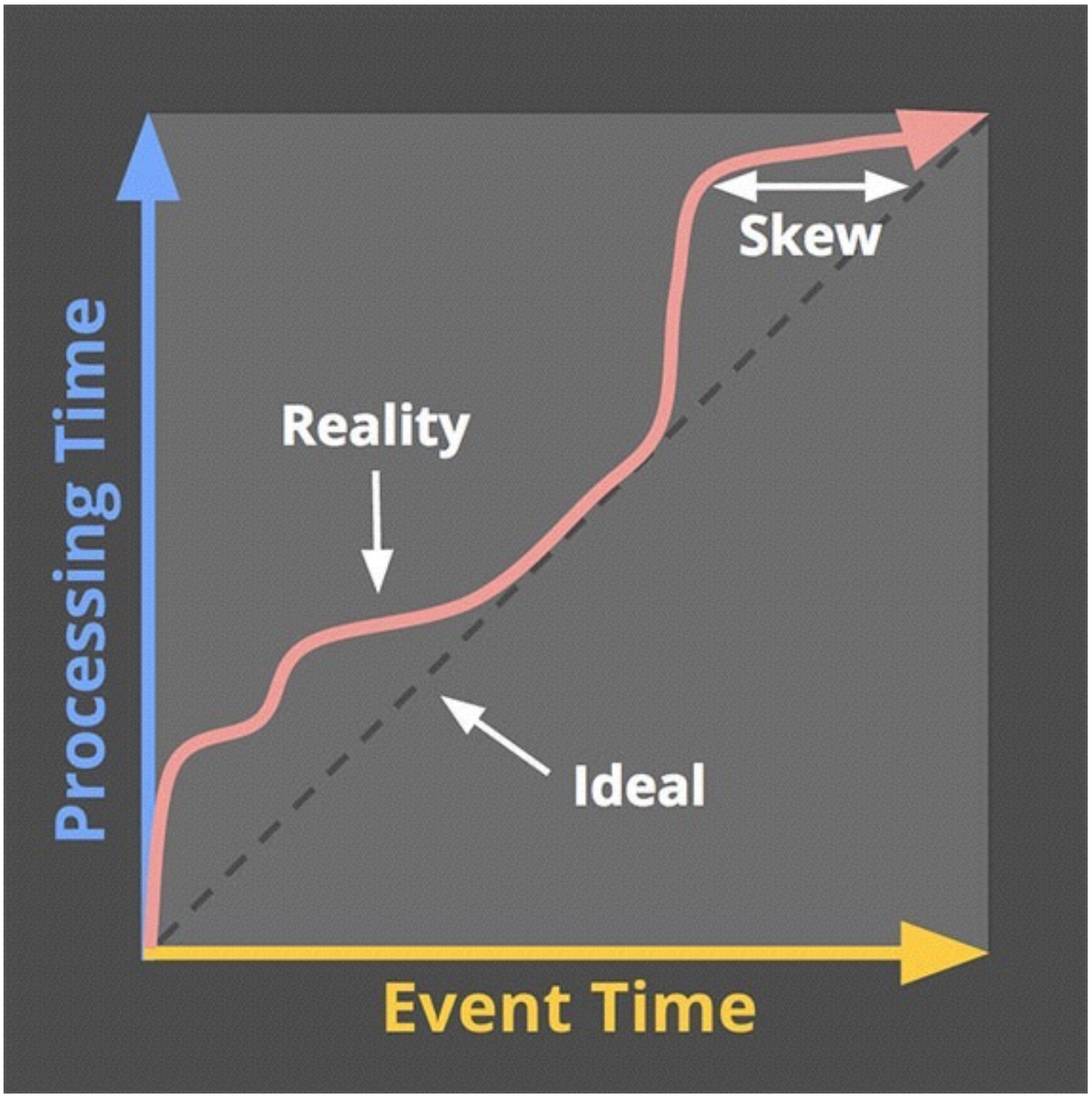


Example: YouTube viewing sessions

Time domains

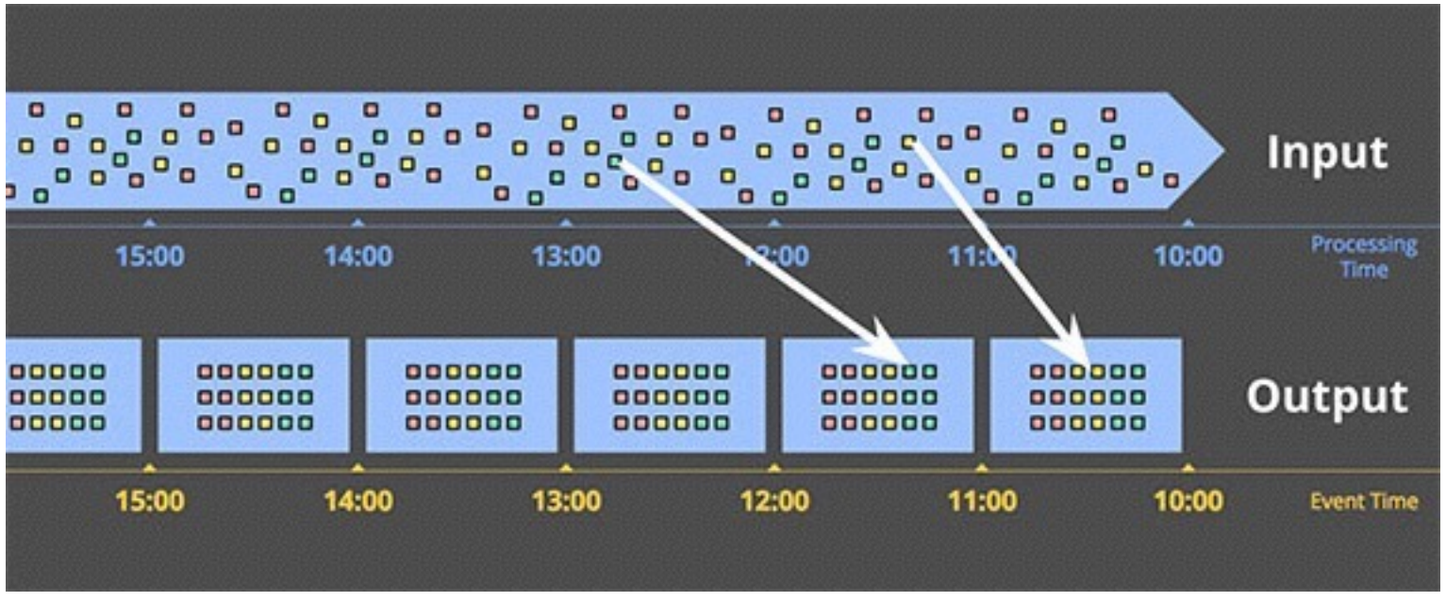
- For many applications, windows should be based on “event time” (when the events actually occur)
  - Example: billing YouTube advertisers
- Lag, partitions, etc, might cause an event to be processed later than its event time
  - Processing time

Challenge: time skew

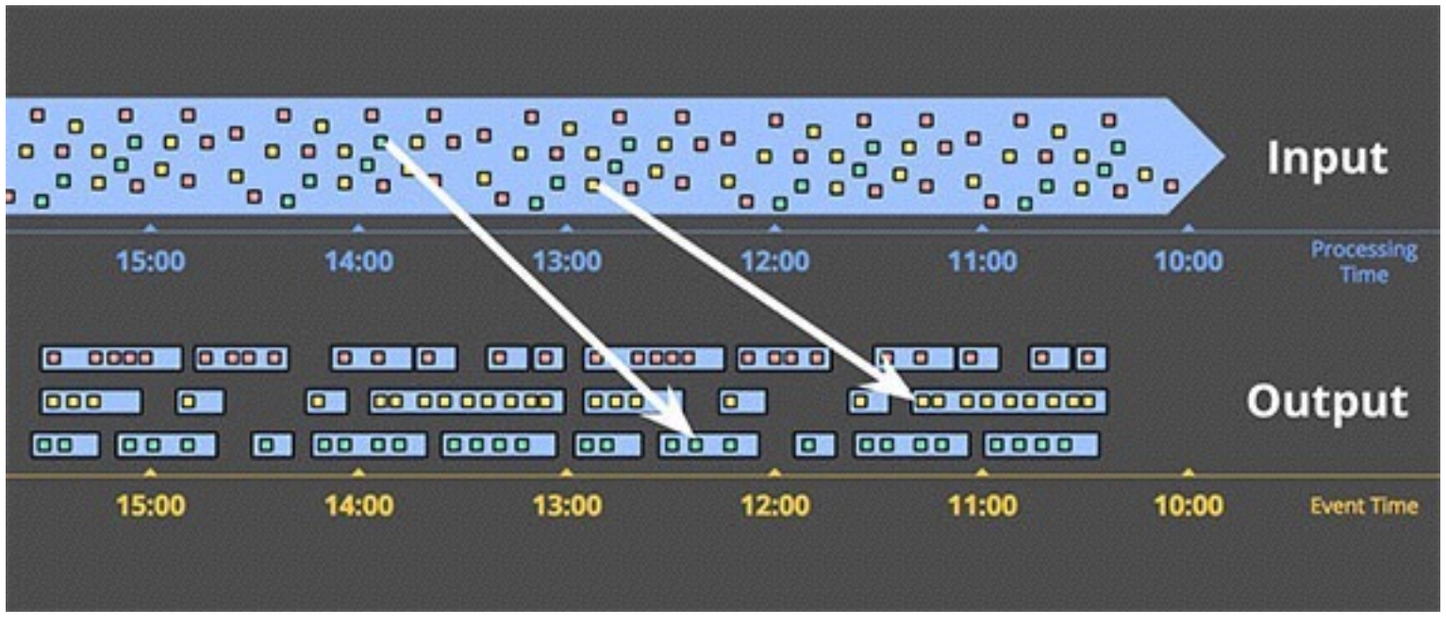


Goal: Event-time windows

Fixed windows



Session windows



Challenge: completion

- With event times, how does the system know if it has received all of the data in a window?

- Example: phones might watch YouTube videos (and ads) offline

Watermarks

- Heuristics that tell the system when it is likely to have received most of the data in a window

- Based on global progress metrics

- Watermarks are insufficient:

  - Late data might arrive behind the watermark

  - Watermark might be too slow due to one late datum and increase latency for the whole system

Incremental processing

- Difficult to get the single best result from a window

- Instead, let windows produce multiple results (improving incrementally over time)

Triggers

- Triggers specify when to output window results
  - at watermark
  - at percentile watermark
  - every minute, etc
- Triggers specify how to output results
  - discard previous window
  - accumulate
  - accumulate and retract
- Triggers are composable

Examples

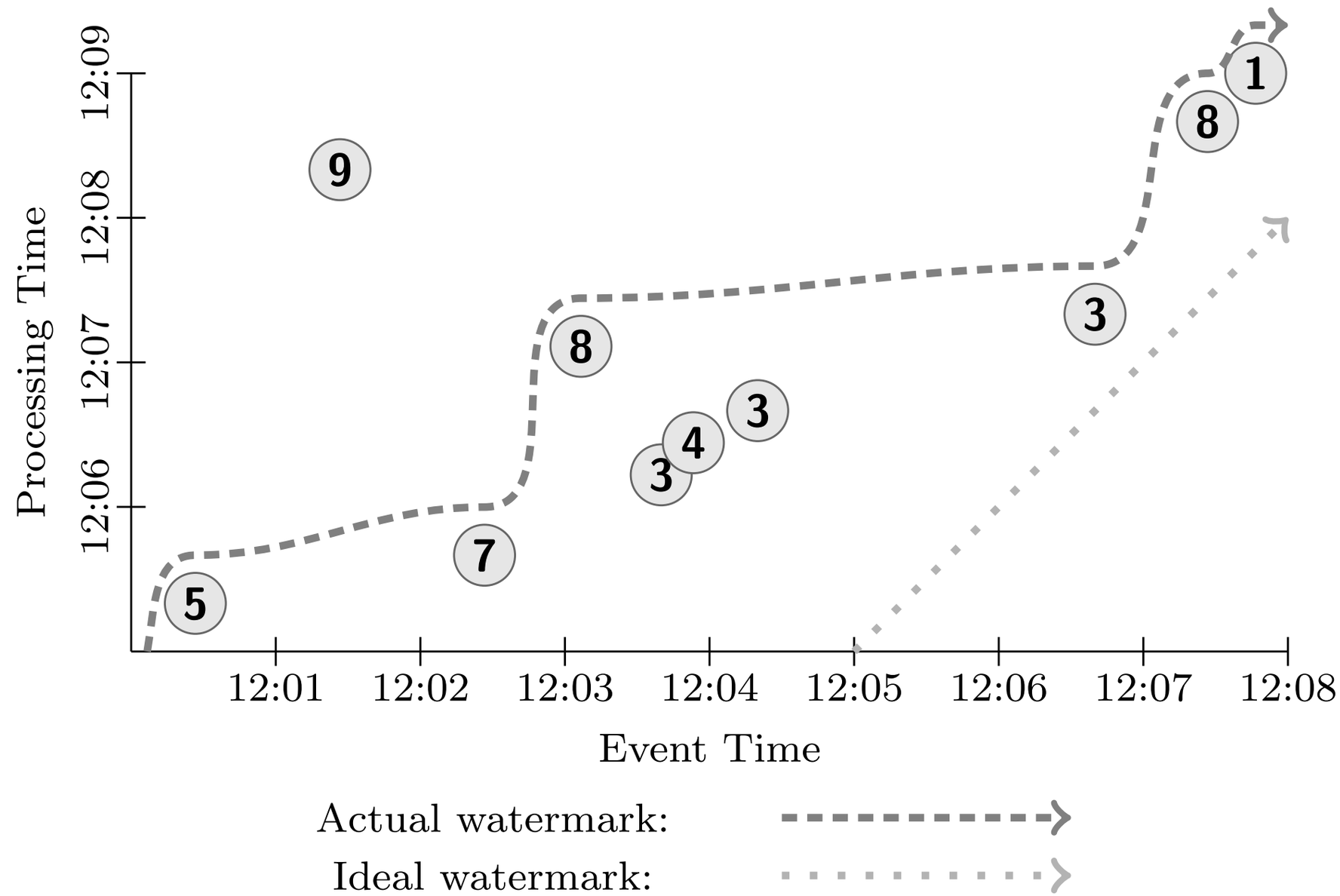


Figure 5: Example Inputs

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
                .accumulating())
  .apply(Sum.integersPerKey());
```

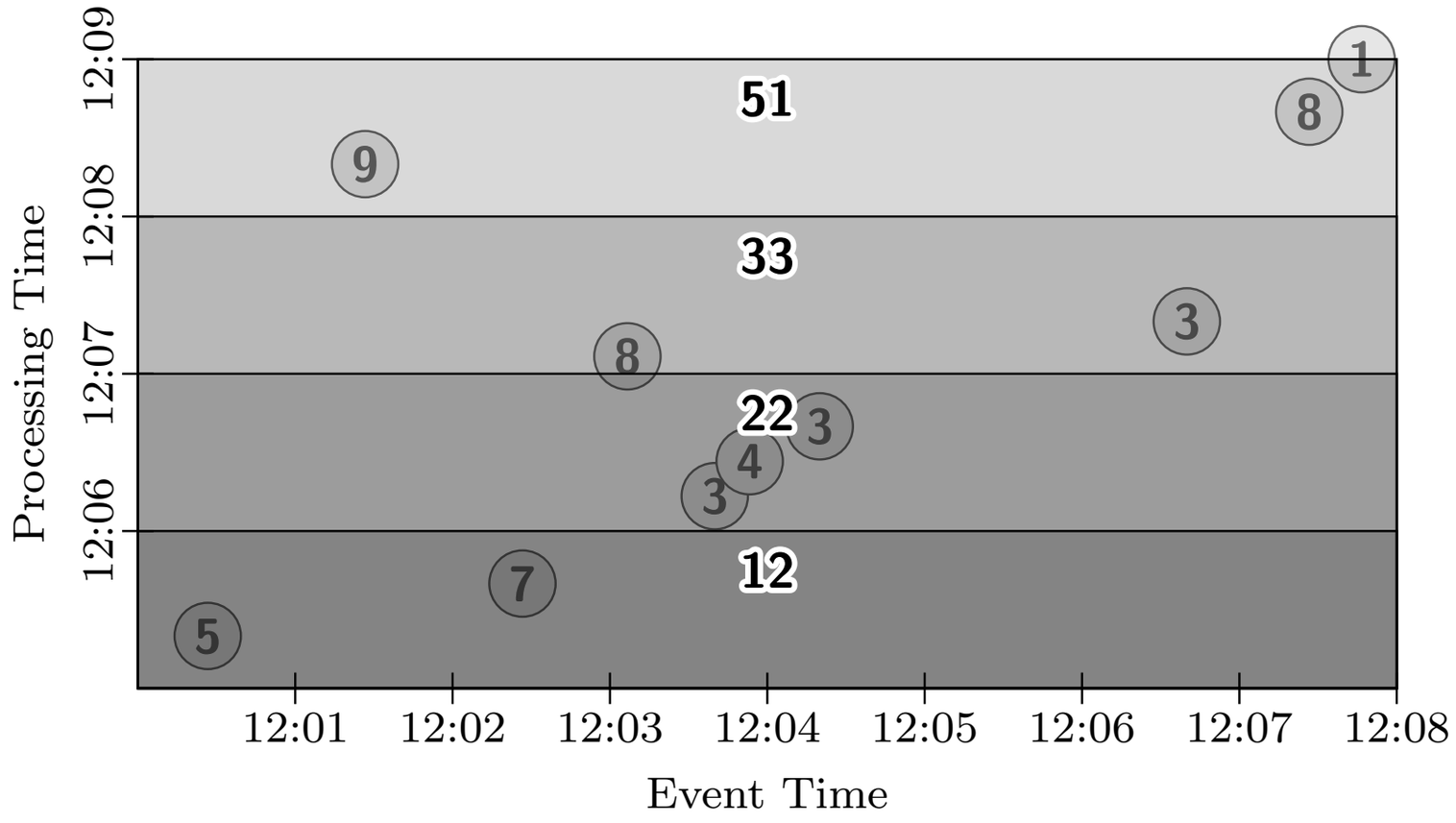


Figure 7: GlobalWindows, AtPeriod, Accumulating

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.trigger(Repeat(AtPeriod(1, MINUTE)))
                .discarding())
  .apply(Sum.integersPerKey());
```

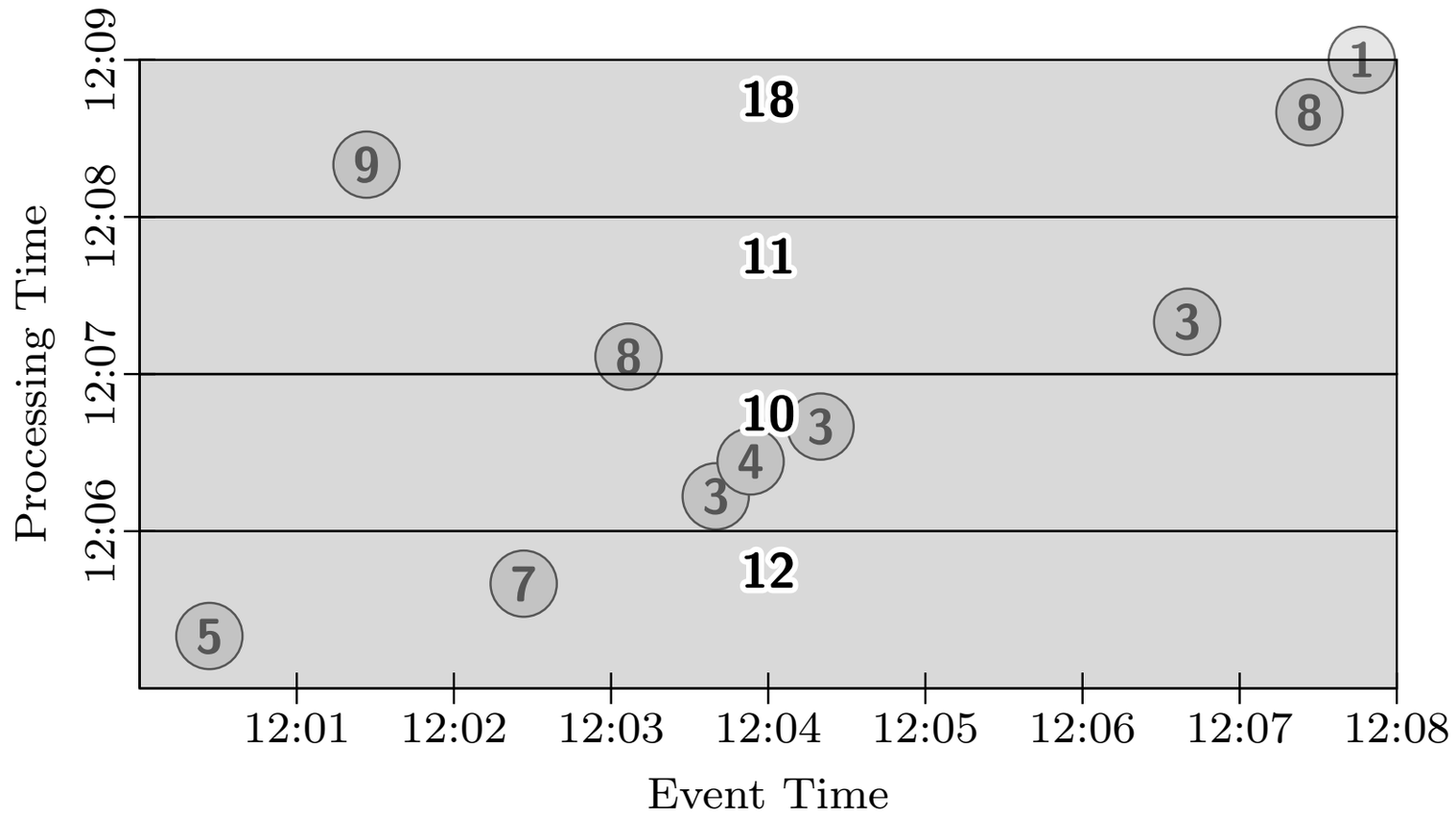


Figure 8: GlobalWindows, AtPeriod, Discarding

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(FixedWindows.of(2, MINUTES))
                .trigger(Repeat(AtWatermark())))
                .accumulating())
  .apply(Sum.integersPerKey());
```

Let’s run this pipeline under the three execution engines: batch, micro-batch, streaming

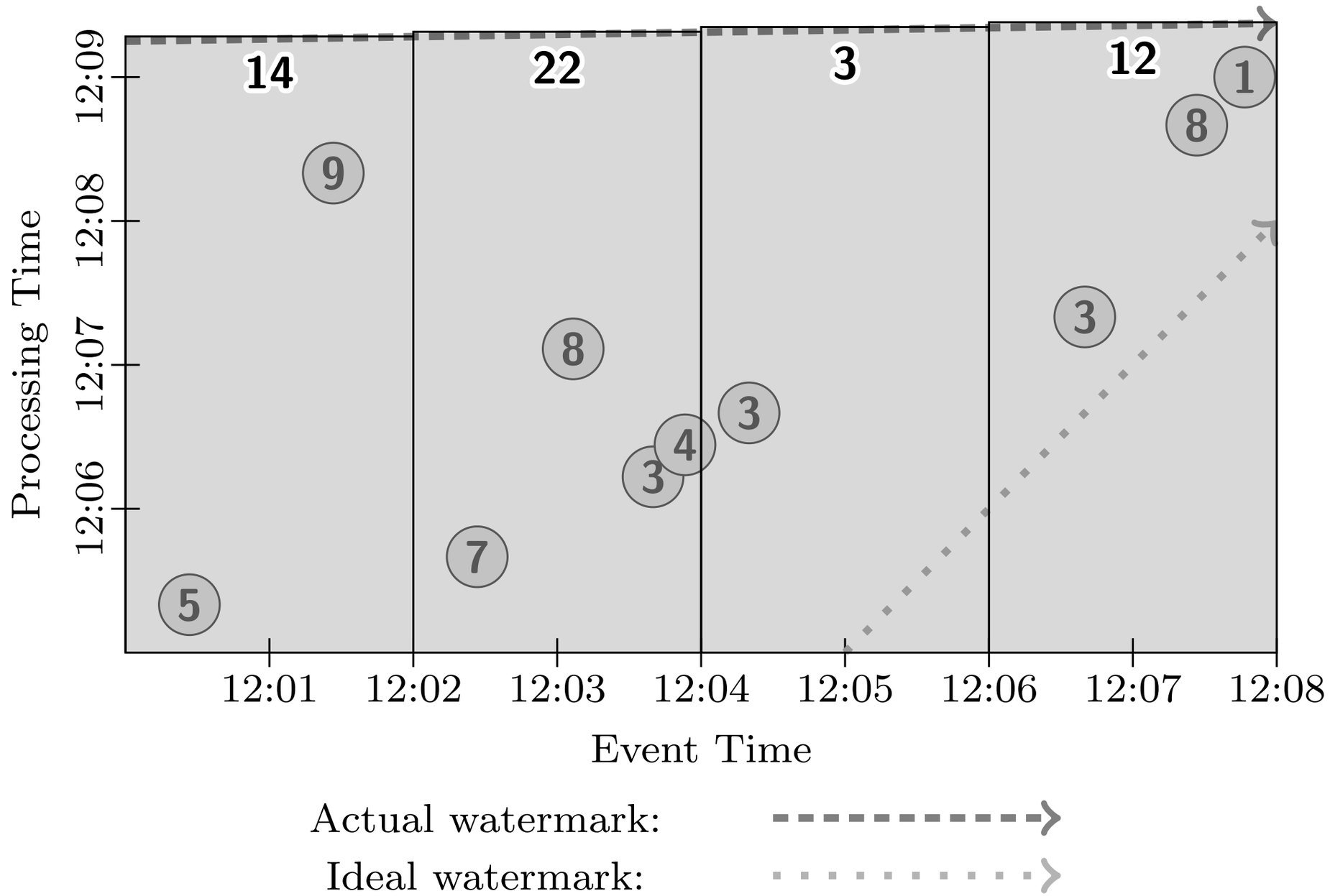


Figure 10: FixedWindows, Batch

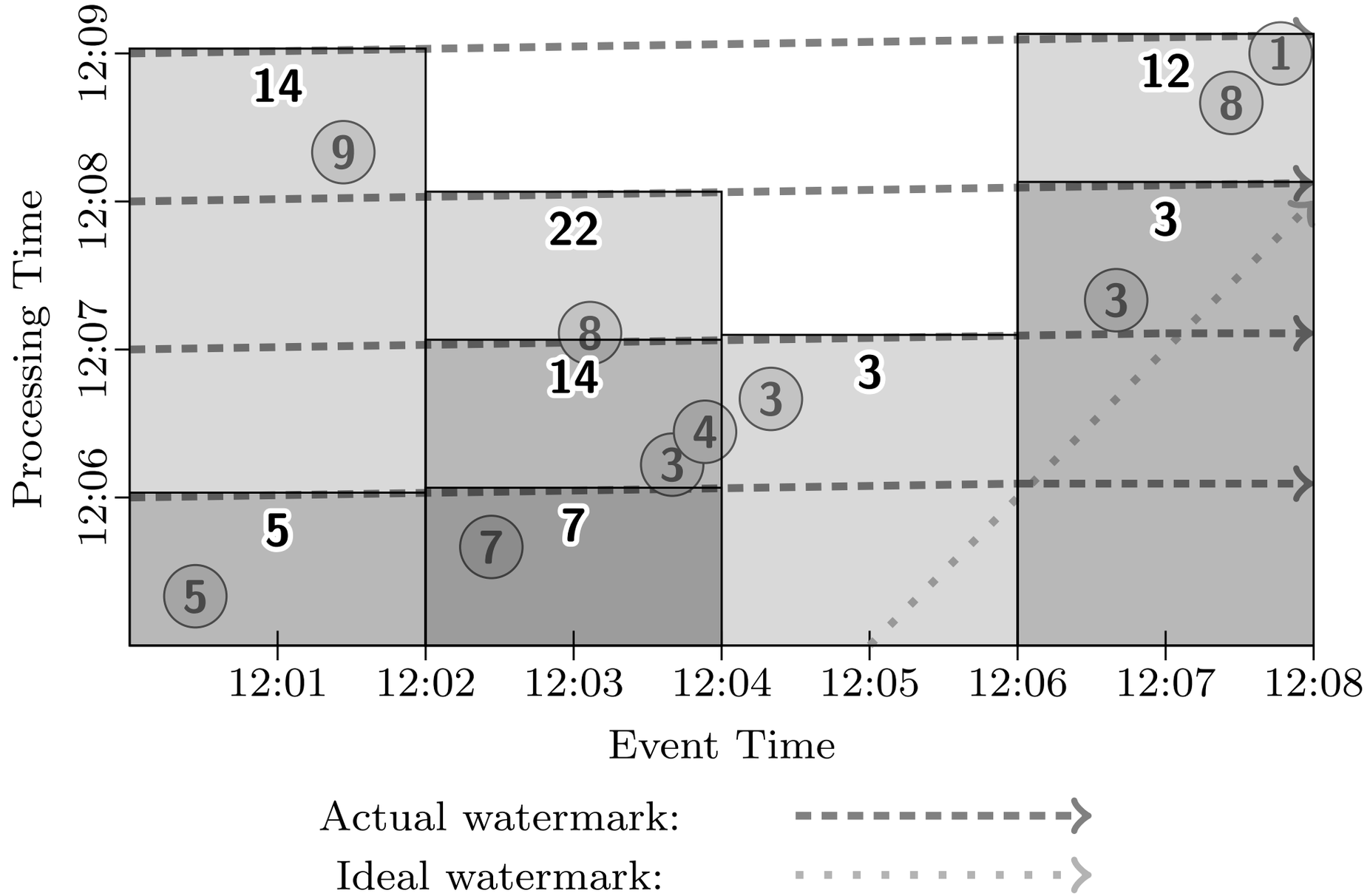


Figure 11: FixedWindows, Micro-Batch

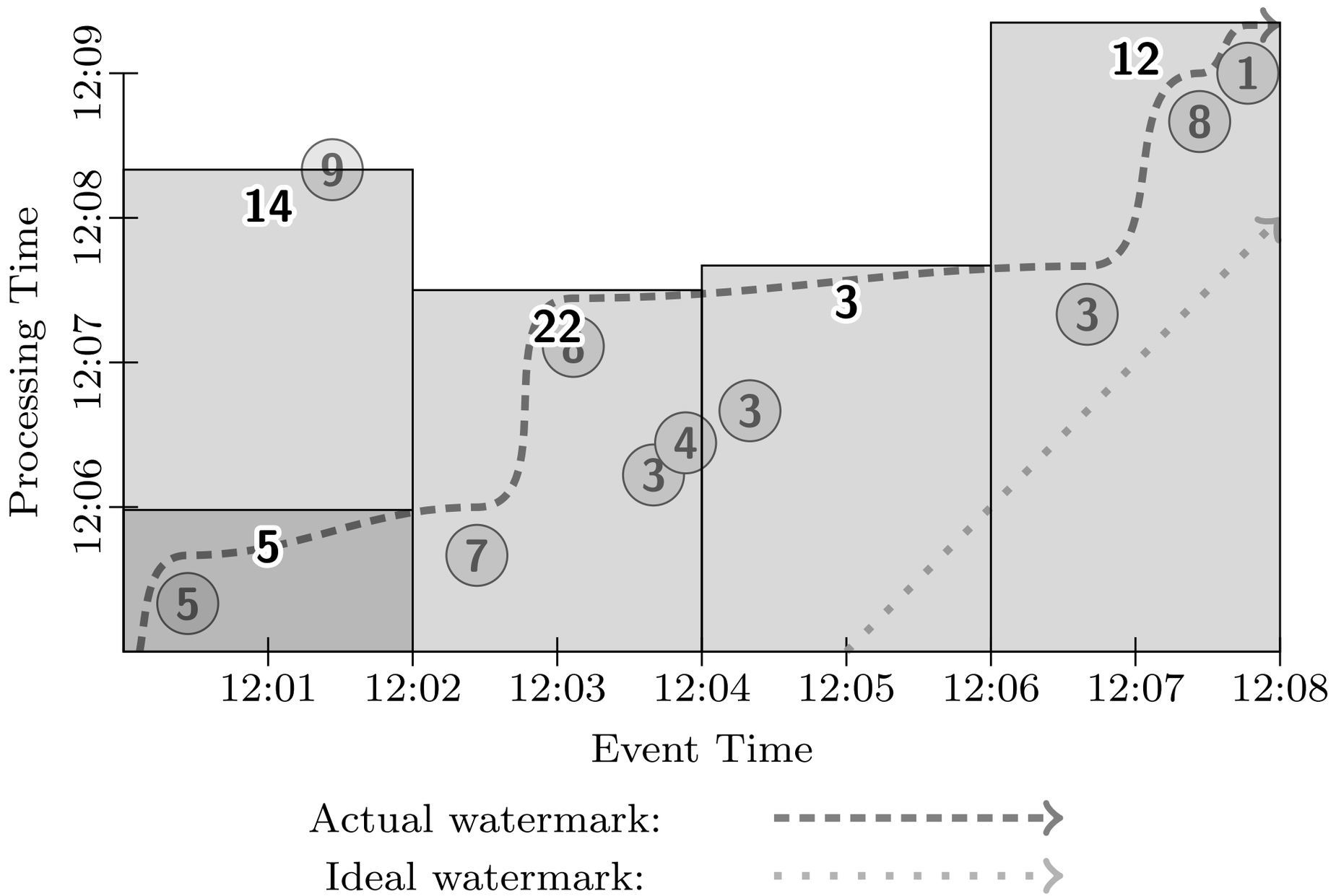


Figure 12: FixedWindows, Streaming

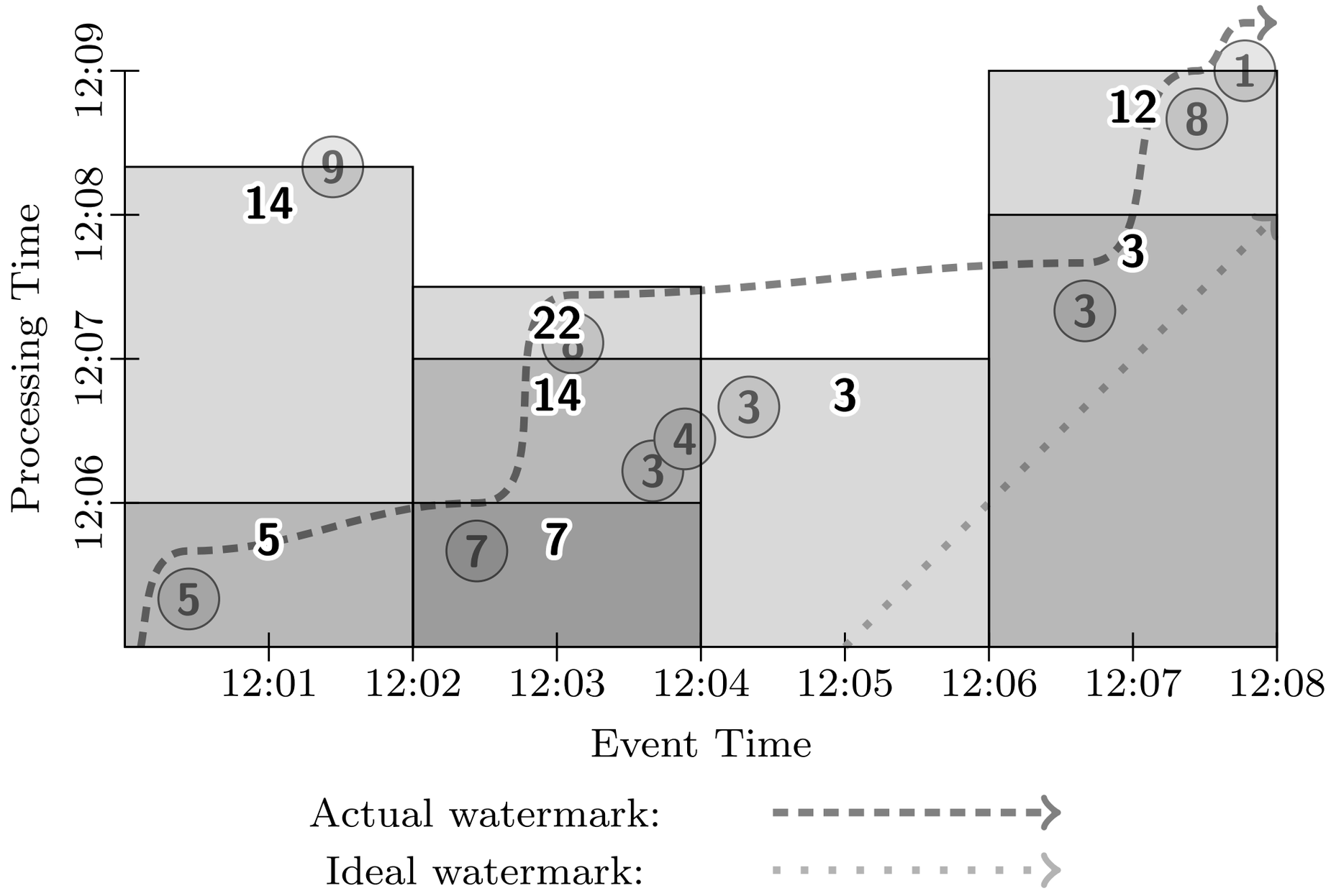


Figure 13: FixedWindows, Streaming, Partial

```
PCollection<KV<String, Integer>> output = input
  .apply(Window.into(Sessions.withGapDuration(1, MINUTE))
                .trigger(SequenceOf(
                  RepeatUntil(
                    AtPeriod(1, MINUTE),
                    AtWatermark()),
                  Repeat(AtWatermark())))
                .accumulatingAndRetracting())
  .apply(Sum.integersPerKey());
```

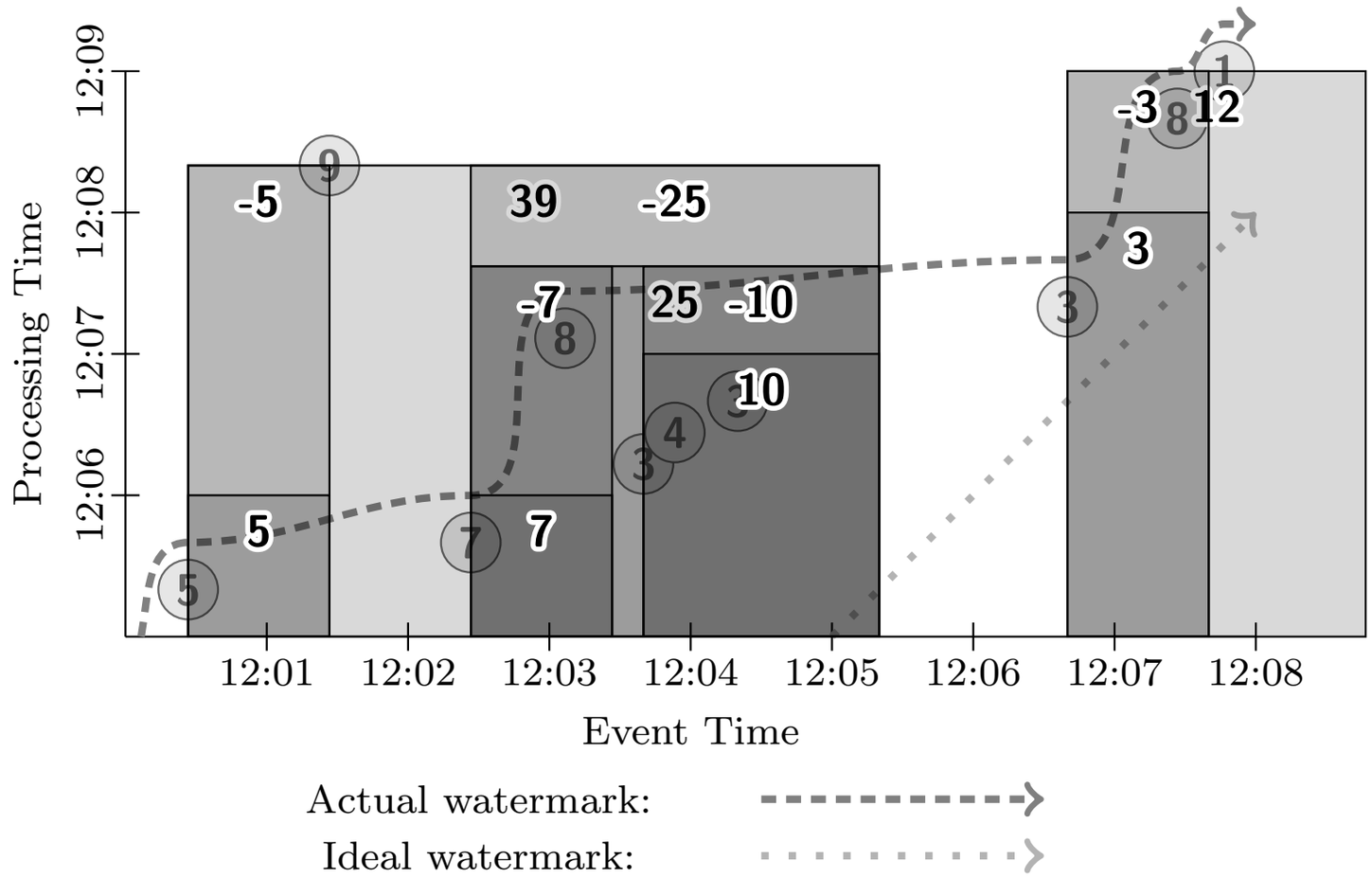


Figure 14: Sessions, Retracting