Streaming Systems

Tyler Akidau & Slava Chernyak & Reuven Lax

June 11, 2023

Contents

1 Streaming 101

A type of data processing engine that is designed with infinite datasets in mind.

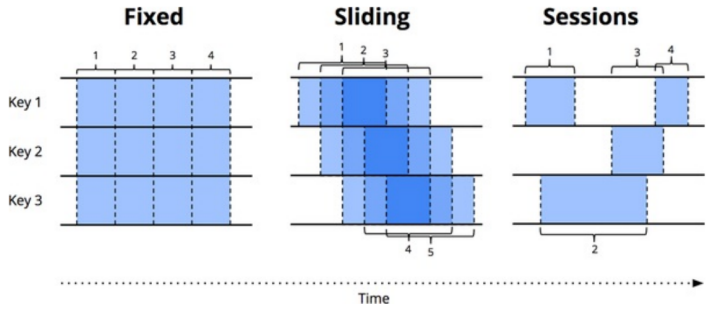
- Correctness: Streaming systems need a method for checkpointing persistent state over time. [**?**], spark streaming: [**?**], flink snapshot: [**?**]
- Tools for reasoning about time.

Event time: when events actually occurred
Processing time: when events are observed in the system

- shared resource limitations, like network congestion, network partition, shared CPU in a nondedicated environment
- software
- features of the data

Fixed windows:

- Problem: network partition, events are collected globally and must be transferred to a common location before processing, delaying processing until you're sure all events have been collected or reprocessing the entire batch for a given window whenever data arrive late
- Time-agnostic:
  - filtering
  - inner joins
- Approximation algorithms: top-n, k-means
- windowing



- Windowing by processing time: the system essentially buffers up incoming data into windows until some amount of processing time has passed

  Problem: if the data in question have event times associated with them, those data must arrive in event-time order if the processing-time windows are to reflect the reality of when those events actually happened.
- windowing by event time

2 The What, Where, When, and How of Data Processing

- What results are calculated?
- Where in event time are results calculated?
- When in processing time are results materialized?
- How do refinements of results relate?

2.1 What: Transformations

In the rest of this chapter (and indeed, through much of the book), we look at a single example: computing keyed integer sums over a simple dataset consisting of nine values.

Name	Team	Score	EventTime	ProcTime
Julie	TeamX	5	12:00:26	12:05:19
Frank	TeamX	9	12:01:26	12:08:19
Ed	TeamX	7	12:02:26	12:05:39
Julie	TeamX	8	12:03:06	12:07:06
Amy	TeamX	3	12:03:39	12:06:13
Fred	TeamX	4	12:04:19	12:06:39
Naomi	TeamX	3	12:06:39	12:07:19
Becky	TeamX	8	12:07:26	12:08:39
Naomi	TeamX	1	12:07:46	12:09:00

In Beam:

- `PCollections`: datasets accoss which parallel transformations can be performed

- `PTransforms`: applied to `PCollections` to create new `PCollections`.

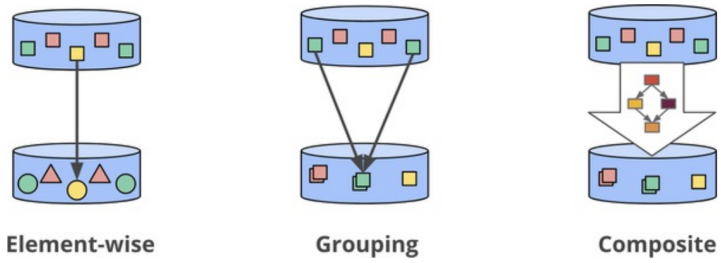


Figure 1: Types of transformations

In our examples, we start out with a pre-loaded `PCollection<KV<Team,Integer>>` named "input"

```
PCollection<String> raw = IO.read(...);
PCollection<KV<Team, Integer>> input = raw.apply(new ParseFn());
PCollection<KV<Team, Integer>> totals =
input.apply(Sum.integersPerKey());
```

For classical batch processing, it looks like this.

2.2 Where: Windowing

```
PCollection<KV<Team, Integer>> totals = input
    .apply(Window.into(FixedWindows.of(TWO_MINUTES)))
    .apply(Sum.integersPerKey());
```

result

As before, inputs are accumulated in state until they are entirely consumed, after which output is produced. In this case, however, instead of one output, we get four: a single output, for each of the four relevant two-minute event- time windows.

2.3 Going Streaming: When and How

2.3.1 When: The Wonderful Thing About Triggers Is Triggers Are Wonderful Things

Two types of triggers:

- **Repeated update triggers**: These periodically generate updated panes for a window as its contents evolve. These updates can be materialized with every new record, or they can happen after some processing-time delay, such as once a minute. The choice of period for a repeated update trigger is primarily an exercise in balancing latency and cost.
- **Completeness triggers**: These materialize a pane for a window only after the input for that window is believed to be complete to some threshold. This type of trigger is most analogous to what we're familiar with in batch processing: only after the input is complete do we provide a result. The difference in the trigger-based approach is that the notion of completeness is scoped to the context of a single window, rather than always being bound to the completeness of the entire input.

```
PCollection<KV<Team, Integer>> totals = input
.apply(Window.into(FixedWindows.of(TWO_MINUTES))
.triggering(Repeatedly(AfterCount(1))));
.apply(Sum.integersPerKey());
```

result

Two approaches for processing-time delays in triggers:

- **aligned delays**: the delay slices up processing time into fixed regions that align across keys and windows
- **unaligned delays**: the delay is relative to the data observed within a given window

```
PCollection<KV<Team, Integer>> totals = input
    .apply(Window.into(FixedWindows.of(TWO_MINUTES))
                 .triggering(Repeatedly(AlignedDelay(TWO_MINUTES)))
    .apply(Sum.integersPerKey());
```

result

The nice thing about it is predictability; you get regular updates across all modified windows at the same time. That's also the downside: all updates happen at once, which results in bursty workloads that often require greater peak provisioning to properly handle the load.

```
PCollection<KV<Team, Integer>> totals = input
    .apply(Window.into(FixedWindows.of(TWO_MINUTES))
                 .triggering(Repeatedly(UnalignedDelay(TWO_MINUTES))
    .apply(Sum.integersPerKey());
```

Listing 1: Triggering on unaligned two-minute processing-time boundaries

result

2.3.2 When: Watermarks

Watermarks are a supporting aspect of the answer to the question: "When in processing time are results materialized?" Watermarks are temporal notions of input completeness in the event-time domain. Worded differently, they are the way the system measures progress and completeness relative to the event times of the records being processed in a stream of events

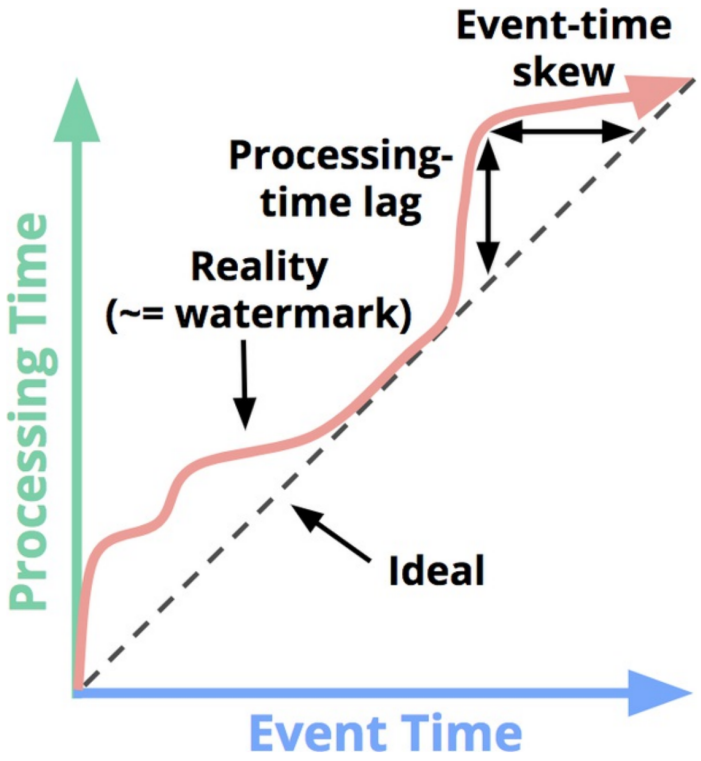


Figure 2: Event-time progress, skew, and watermarks

We can think of the watermark as a function $P \rightarrow F$ from processing time to event time. That point in event time, E, is the point up to which the system believes all inputs with event times less than E have been observed.

Depending upon the type of watermark, perfect or heuristic, that assertion can be a strict guarantee or an educated guess, respectively:

- **Perfect watermarks**: For the case in which we have perfect knowledge of all of the input data, it's possible to construct a perfect watermark.
- **Heuristic watermarks**: use whatever information is available about the inputs (partitions, ordering within partitions if any, growth rates of files, etc.) to provide an estimate of progress that is as accurate as possible. In many cases, such watermarks can be remarkably accurate in their predictions.

Because they provide a notion of completeness relative to our inputs, watermarks form the foundation for the second type of trigger mentioned previously: **completeness triggers**.

```
awefawef
```

Listing 2: test