Making the Jump From Batch To Streaming: Motivations and Concepts

Yi Hu

Before we begin…

This talk goes into the general data processing concepts for streaming.

(i.e. won't touch on Beam-specific implementations and interactions with these concepts)

If you're already familiar with these concepts, and just want to see how to use them in Beam, watch "Making the Jump From Batch to Streaming: Beam Primitives"

Why Streaming?

Why would I need a streaming pipeline?

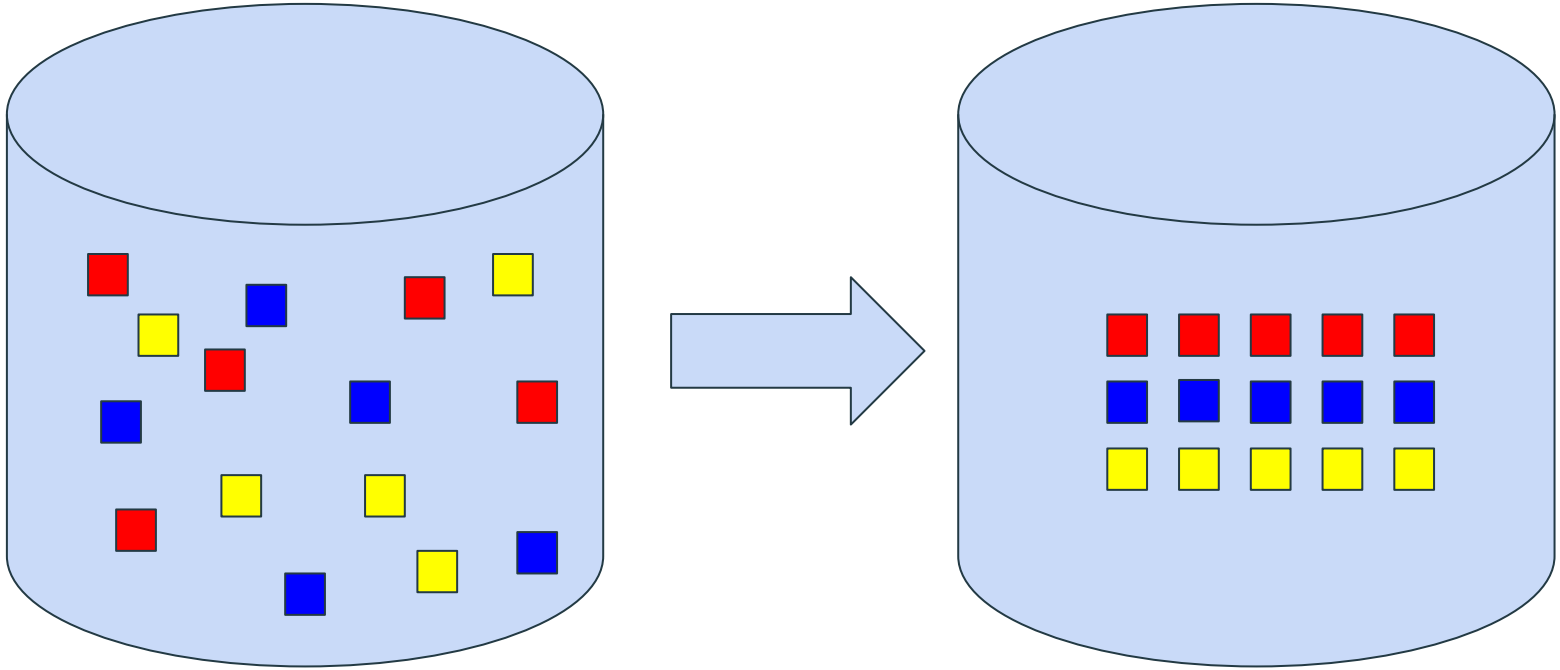
- Want to process your data in real time
  - Data from streaming **source**, e.g. message system
  - Requirement to see **results** in real time
- Have a batch pipeline found running more and more frequently
  - Scheduler + Cron batch job vs single streaming job
  - Batch job regularly cause usage hike to external resources

Differences from Batch

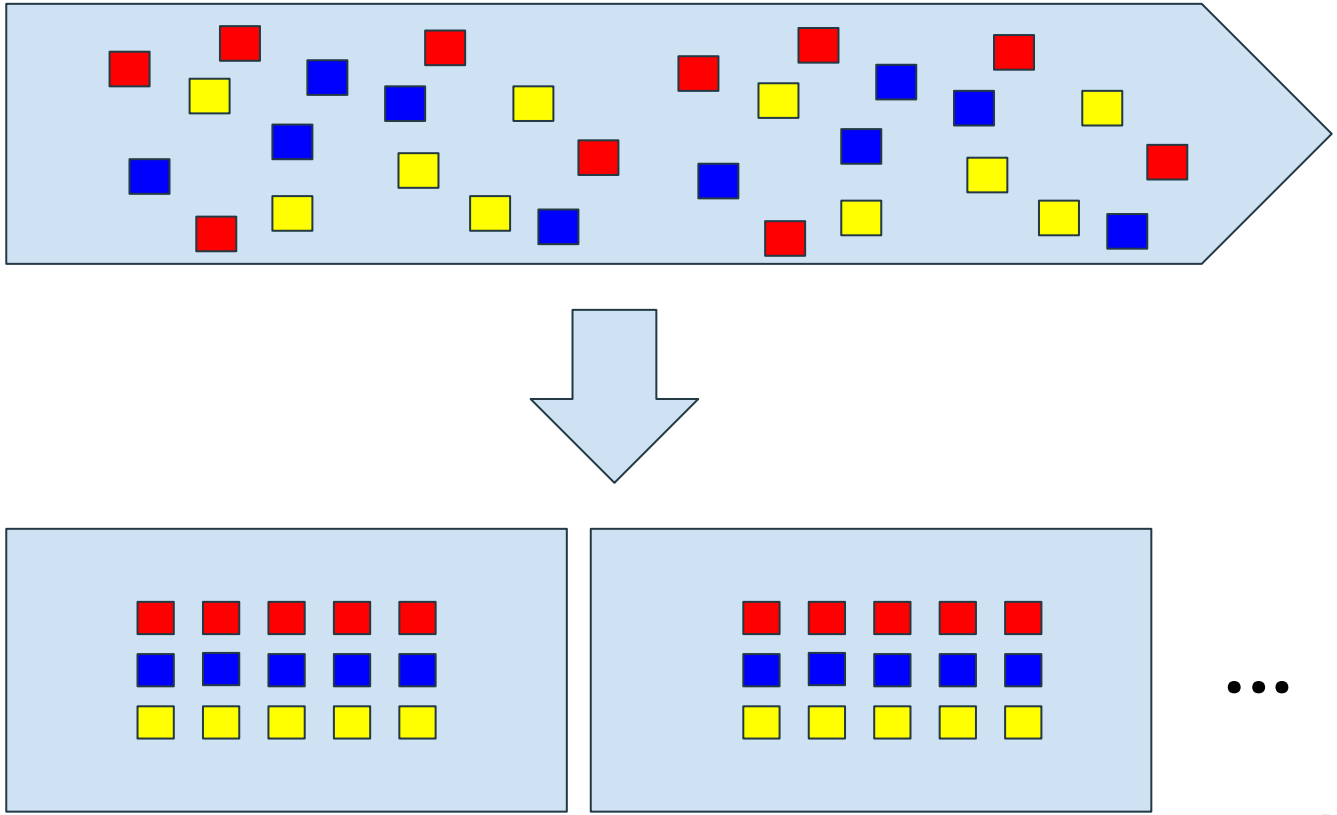
Bounded vs. Unbounded Sources

- Batch pipelines operate using bounded data sources
  - You know *exactly* how much data you have to process at the start of the pipeline
  - Parallelizing that pipeline is a matter of dividing a fixed amount of work
- Streaming pipelines ingest data from unbounded sources
  - The pipeline can read a theoretically infinite amount of data
  - Data can arrive out of order, late, etc.

Bounded Pipelines



Unbounded Pipelines



Processing

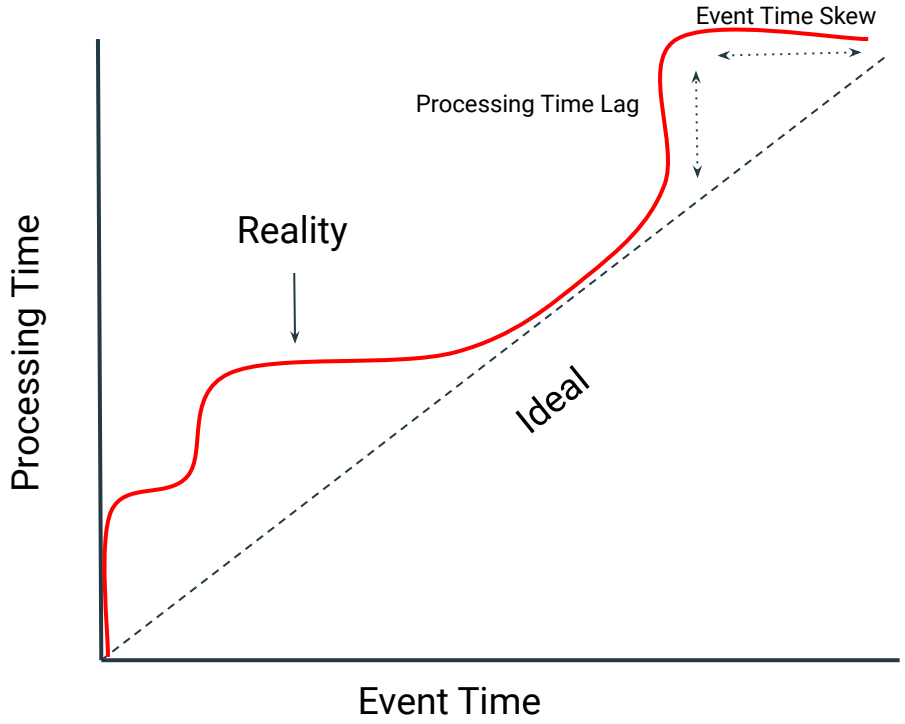
- Batch pipelines have steady data throughput
  - Consistent, large bundle sizes going through DoFns
- Streaming pipelines can have variable data throughput
  - Can have periods of high or low data entering the pipeline
  - This can lead to varying computational needs over the life of a pipeline
  - Bundles are expected to be 1-2 elements at most

Processing in Time

Event Time vs. Processing Time

- Event Time - when the data being processed was created
- Processing Time - when the data is being processed in the pipeline
- Ideally we want these to be the same time for each, but there will always be some delta.

Event Time vs. Processing Time



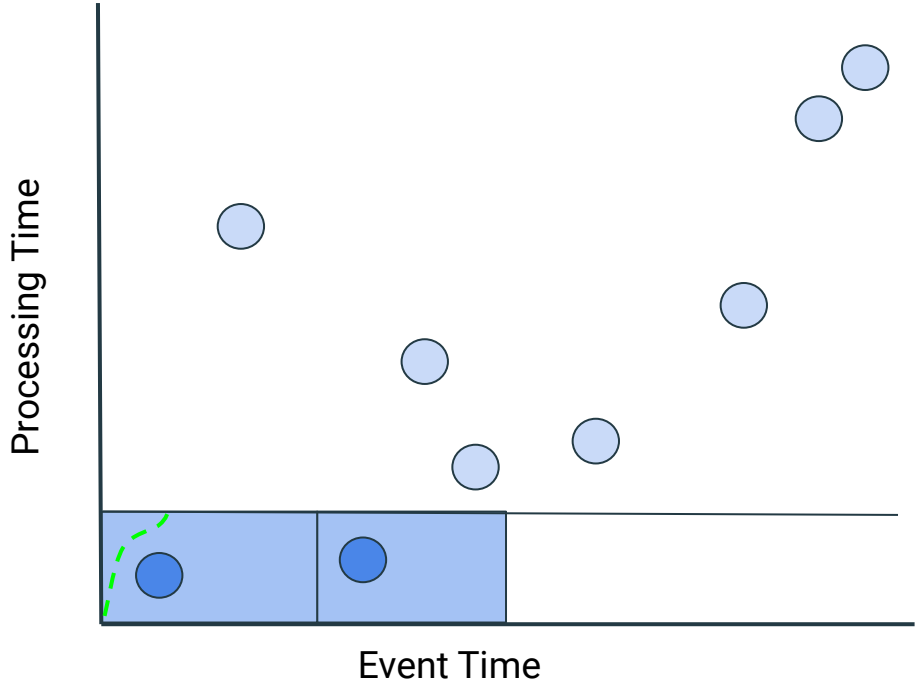
Windows

- Windowing is the process of slicing up a data source based on time
  - This is generally done based on event time, not processing time
- There are multiple approaches that can be taken for windowing
  - Global Windowing
  - Interval Windows
    - Fixed
    - Sliding
  - Session Windows

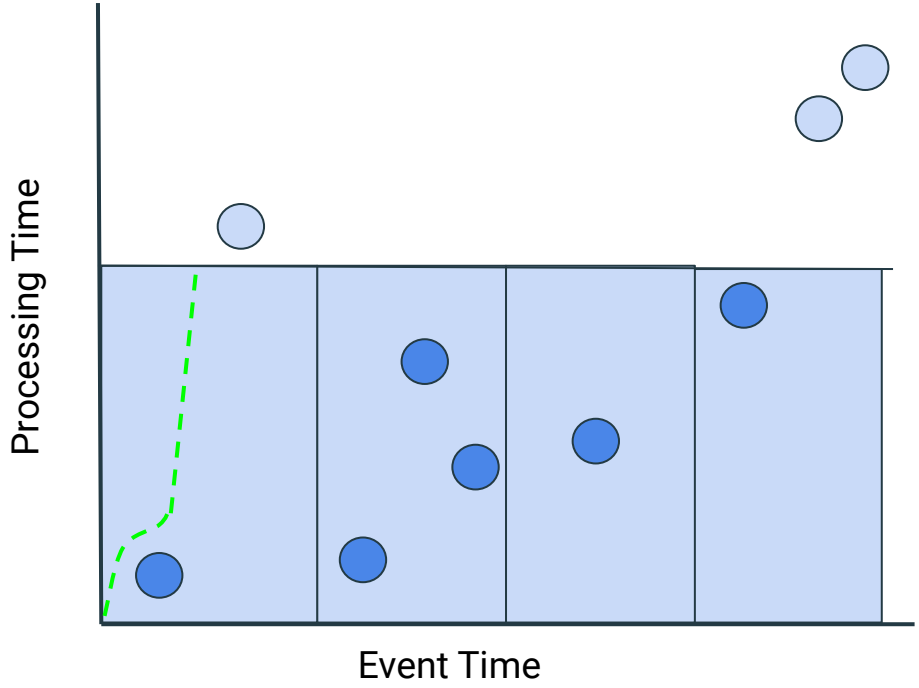
Watermarks

- So we have windows and data that goes in those windows… How do we know when we have all of the data for a given window?
  - Short answer: we cannot know for sure.
- The watermark is how we represent the latest event time for which we *believe* we've ingested all of the data for
  - Mechanically, the system thinks that it will no longer get any data from before the watermark time.

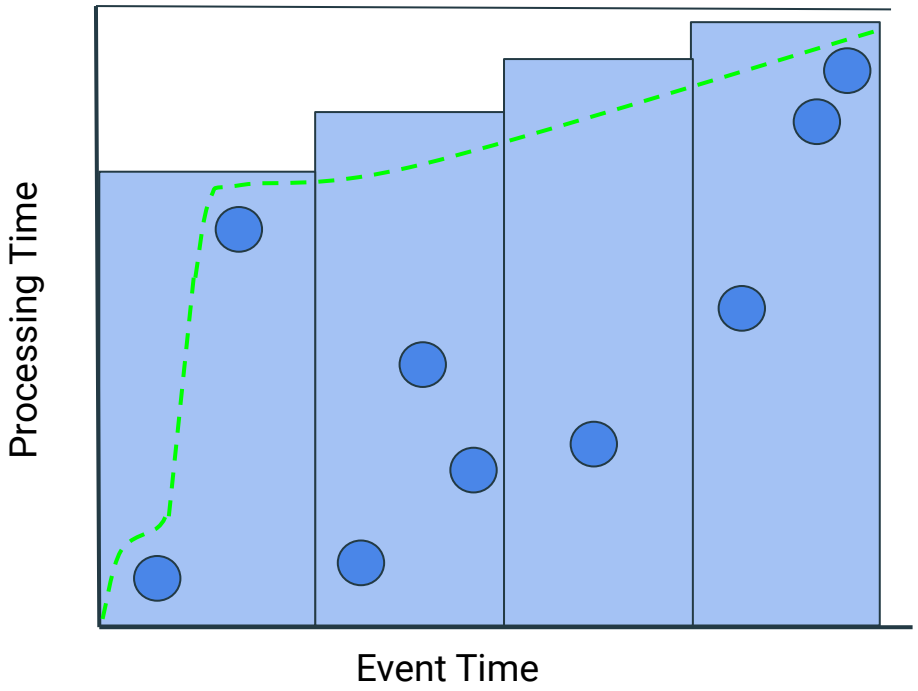
Perfect Watermarks



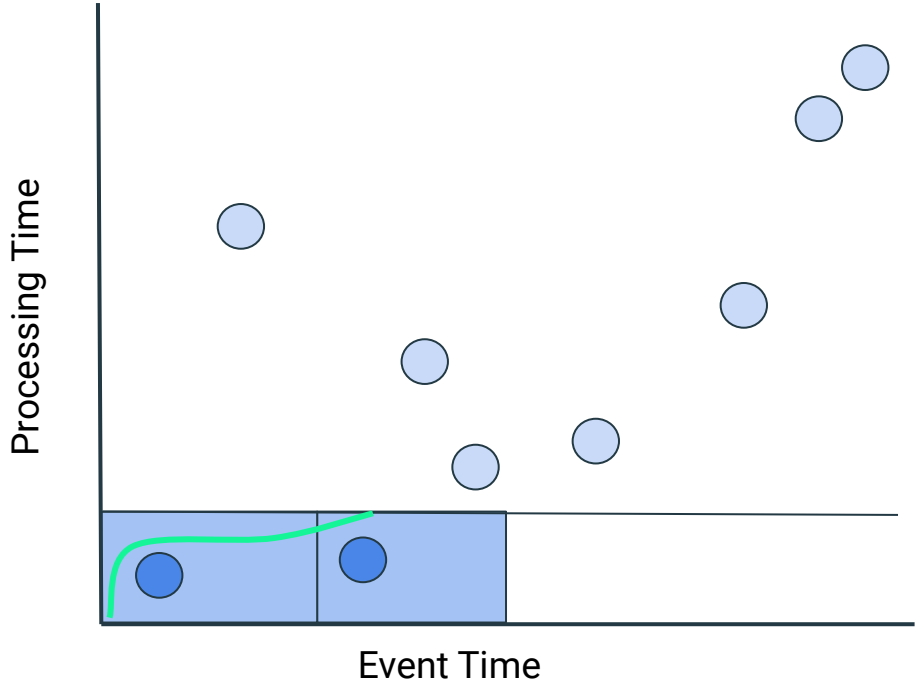
Perfect Watermarks



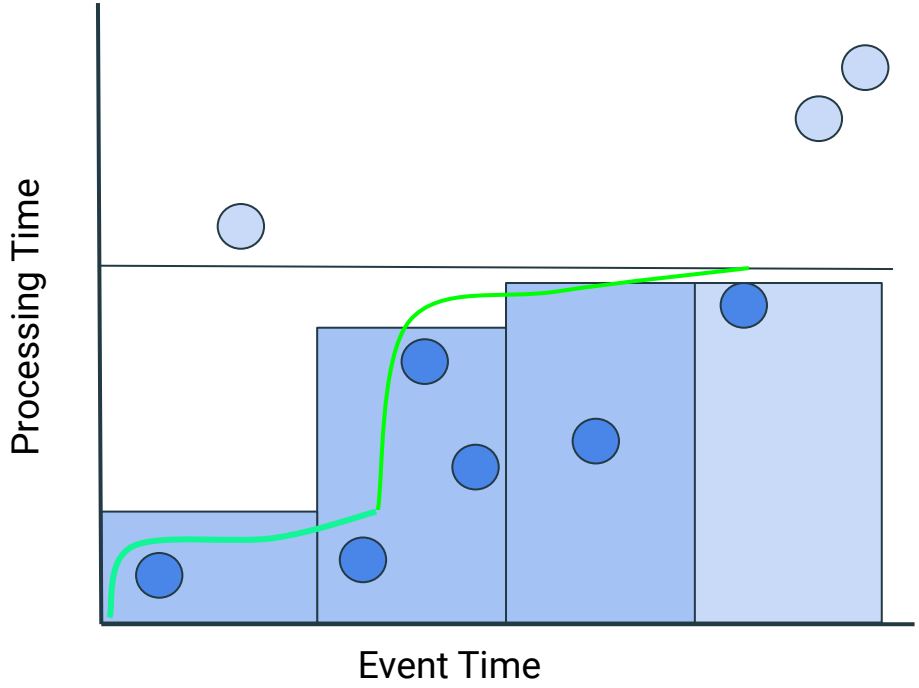
Perfect Watermarks



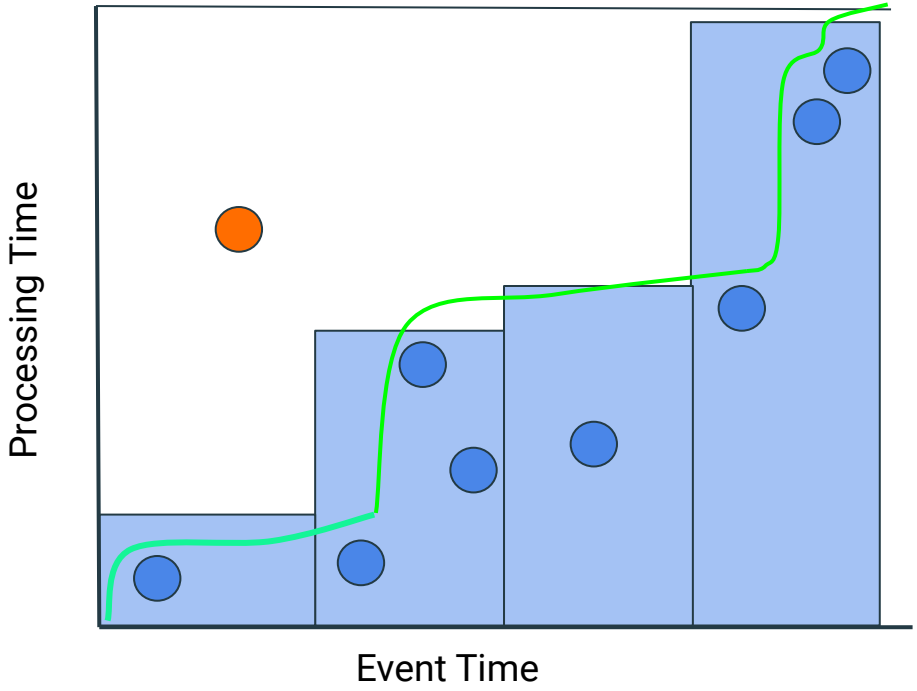
Heuristic Watermarks



Heuristic Watermarks



Heuristic Watermarks



Late Data

- But what happens when we *do* get late data (AKA data from before the current watermark)?
  - The real world is messy, data can be delayed for a number of reasons
- Our pipeline needs some rules on how to handle late data
  - How late will we allow data to be and still be included, if at all?

On the next talk…

We'll get into how you manage these primitives and understand these concepts in Beam, with a few examples and explanations.

Extra Reading

- Streaming Systems: The What, Where, When, and How of Large Scale Data Processing by Tyler Akidau, Slava Chernyak, and Reuven Lax
  - An overview of data processing concepts for both batch and streaming written from the perspective of Apache Beam

Making the Jump From Batch To Streaming: Beam Primitives

Yi Hu

Before we begin…

This talk goes into the specific Beam primitives for streaming pipelines and how they're used.

If you want explanations as to why you would want/need to use these primitives, you should watch "Making the Jump From Batch to Streaming: Motivations and Concepts" before jumping into this content.

Reading Data From Streaming Sources

Splittable DoFns

- Splittable DoFns (SDFs) are transforms that provide information to the runner about the work being done
  - How much work there is to do
  - How much work has already been done
- To do so, an SDF requires
  - RestrictionTracker -- tracks the claimed part of work, used to distribute/split works
  - WatermarkEstimator---estimating output_watermark based on the timestamp of output records or manual modifications

Splittable DoFns

- These allow for work to be rebalanced on the fly based on long-running work items, autoscaling, etc.
- SDFs are extremely important for IOs, and more specifically streaming IOs

Example Beam IO with SDF implementation

- (Java) KafkaIO (ReadFromKafkaDoFn)
- (Python) PeriodicImpulse

Further reading

https://github.com/iht/splittable-dofns-python/ Beam Summit 2022 workshop: Splittable DoFn for Python

UnboundedSource

- Legacy way of reading from unbounded source (Java only)
  - Read(UnboundedSource) transform as a source node of the job graph
  - UnboundedSource.createReader creates an reader that runner uses to read data from source
  - UnboundedReader.start() and .advance() conduct actual read
  - runner gets current element by Reader.getCurrent()
- Executed via wrapper of SDF on portable runners
- Example Beam IO with UnboundedSource implementation
  - most Java SDK streaming sources e.g. JmsIO; SolaceIO (new)

The source tracks:

- Watermark estimation
  - Remember that we must lean on heuristics to estimate watermarks and can never be perfect
- (optional) getBacklogBytes
- (optional) report throttling metrics
  - Metrics.counter("beam-throttling-metrics", "throttling-msecs");

} help runner scaling decision

Elements in processing

"Metadata" associated with elements

- **Key**

- Timestamp
- After Window.into transform applied:
  - Window(s - element can be associated with multiple windows):
- After GroupByKey firing
  - pane (every pane is implicitly associated with a window)

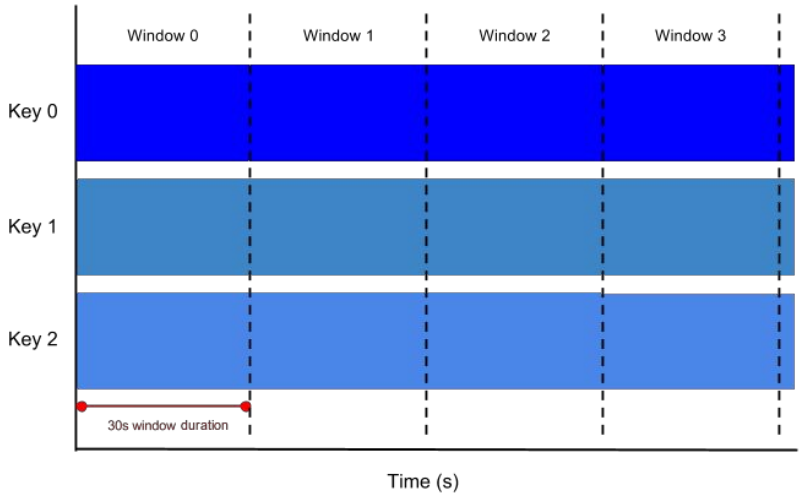
Windowing Strategies

Windowing Types

- The Global Window
  - One large single window, from Unix time 0 to max Unix time
  - By default, Beam operates on the global window until told otherwis
- Interval Windows
  - Fixed Intervals
  - Sliding Windows
- Session Windows

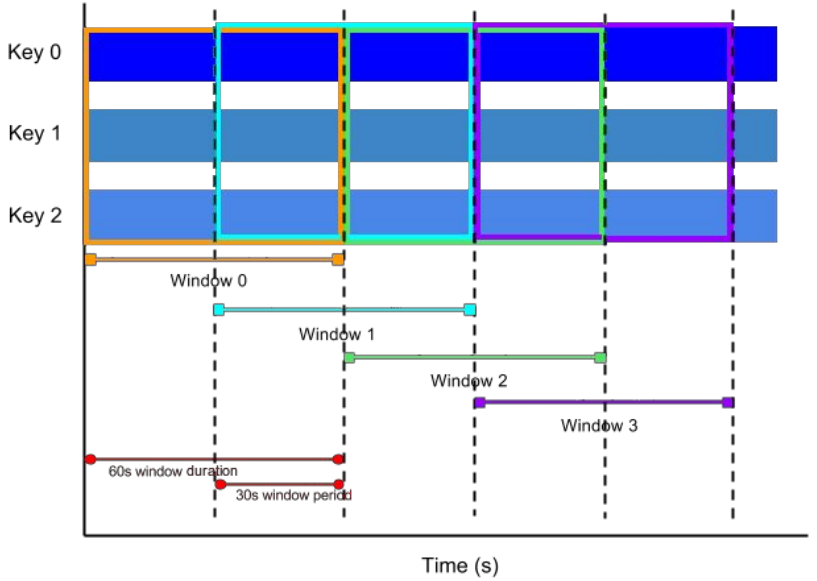
Fixed Interval Windows

- Windows of a fixed length that also occur at that same interval
- No gaps in windows, no overlap between windows




```
PCollection<String> items = ...;
PCollection<String> fixedWindowedItems = items.apply(
 Window.<String>into(FixedWindows.of(Duration.standardSeconds(60))));
```

Sliding Interval Windows

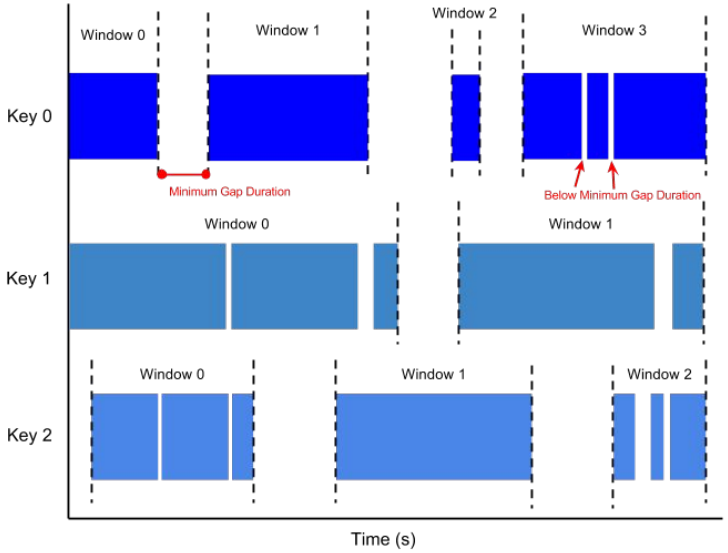


- Windows of a fixed length and fixed period, but the period is less than the length
- The effect is a series of windows with some amount of overlap
  - Data can exist in more than one window at a time

```
PCollection<String> items = ...;
PCollection<String> slidingWindowedItems = items.apply(
   Window.<String>into(
    SlidingWindows.of(Duration.standardSeconds(30)).every(
     Duration.standardSeconds(5))));
```

Session Windows

- A dynamic windowing, windowing events that occur with a length of time shorter than some timeout
- These are “merging” windows, as they can grow and merge with other windows as data is processed




```
PCollection<String> items = ...;
PCollection<String> sessionWindowedItems = items.apply(
   Window.<String>into(Sessions.withGapDuration(
      Duration.standardSeconds(600))));
```

Triggers

What is a Trigger?

- Triggers are a mechanism that declare when output for a given window should be materialized relative to an external signal
- Triggers can also be configured to fire multiple times, allowing for the output for a window to be updated over time as more data comes in
- This is how Beam can handle revisions to data or data that comes in late

Panes

- When a trigger is activated, the materialized output for the given window is commonly referred to as a “pane”
- Multiple panes can be output for a single window as processing continues

Common Trigger Types

- Repeated Update Triggers
  - The output for a window can be repeatedly updated, either with the presence of new records or after some amount of processing time
  - Using these triggers generally requires some balancing of latency and cost depending on how expensive the transform being calculated is
- Completeness Triggers
  - The output is materialized when some threshold of completeness is reached
  - Allow for handling of late/missing data

Code Example

```
// Watermark Completeness Trigger
PCollection<String> items = ...;
PCollection<String> fixedWindowedItems = items.apply(
   Window.<String>into(
      FixedWindows.of(Duration.standardMinutes(1)))
          .triggering(Repeatedly.forever(
              AfterWatermark.pastEndOfWindow()))
          .discardingFiredPanes());

// Repeated Update Trigger
PCollection<String> items = ...;
PCollection<String> globalWindowItems = items.apply(
    Window.<String>into(new GlobalWindows()).triggering(
        Repeatedly.forever(
            AfterPane.elementCountAtLeast(5)))
        .discardingFiredPanes());
```

Managing Panes

Window Accumulation Modes

- When you specify a trigger you must also specify an accumulation mode for the window - what do we do with data in panes that have been emitted before?

Accumulating

- An accumulating trigger re-emits the data that was previously emitted along with the new data
- Set by invoking `.accumulatingFiredPanes()` when setting the trigger
- For example, if we had an accumulating trigger that fired every three elements:

```
First trigger firing:  [5, 8, 3]
Second trigger firing: [5, 8, 3, 15, 19, 23]
Third trigger firing:  [5, 8, 3, 15, 19, 23, 9, 13, 10]
```

Discarding

- A discarding trigger only emits the new data it has received since it last fired
- Typically useful if a downstream transform is doing an aggregation using the panes already
- Set by invoking `.discardingFiredPanes()` when setting the trigger
- Our previous example, but with a discarding trigger:

```
First trigger firing:  [5, 8, 3]
Second trigger firing:          [15, 19, 23]
Third trigger firing:                       [9, 13, 10]
```

Extra Reading

- Streaming Systems: The What, Where, When, and How of Large Scale Data Processing by Tyler Akidau, Slava Chernyak, and Reuven Lax
  - An overview of data processing concepts for both batch and streaming written from the perspective of Apache Beam
- The Beam Programming Guide
  - Free online resource with explanations of these concepts and code snippets per language

Handling Late Data

Late Data Handling

- In the case of completeness triggers, we want to provide some measure of handling for how late we allow data to be
- This is important, as you don't want to have to repeatedly recalculate panes for old data
  - Implications for things like caching and garbage collection in workers

Allowed Lateness

- Enter the “allowed lateness” option
- The allowed lateness is relative *to the current watermark value*
  - Simpler process compared to keeping up with processing time

A More Complex Code Example

```
PCollection<String> items = ...;
PCollection<String> fixedWindowedItems = items.apply(
   Window.<String>into(
      FixedWindows.of(ONE_MINUTE)
      .triggering(
         AfterWatermark()
           .withEarlyFirings(AlignedDelay(ONE_MINUTE))
           .withLateFirings(AfterCount(1))
      )
      .withAllowedLateness(Duration.standardDays(2))
.discardingFiredPanes());
```

Thank you!