Writing a Native Go Streaming Pipeline

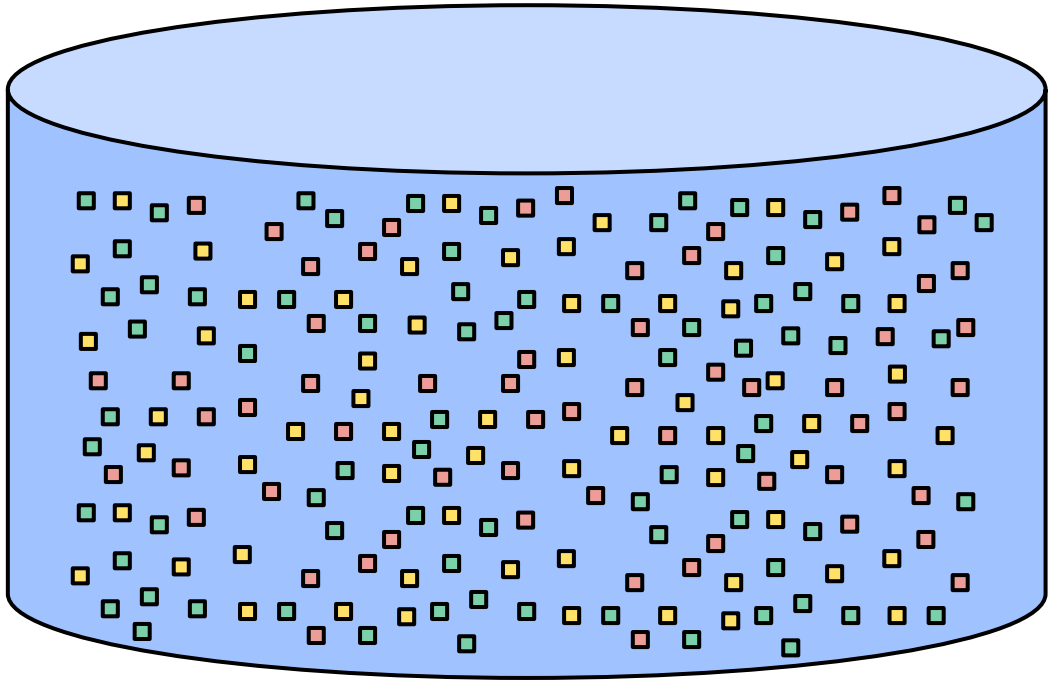
Danny McCormick (@damccorm)
Jack McCluskey (@jrmccluskey)

BEAM SUMMIT

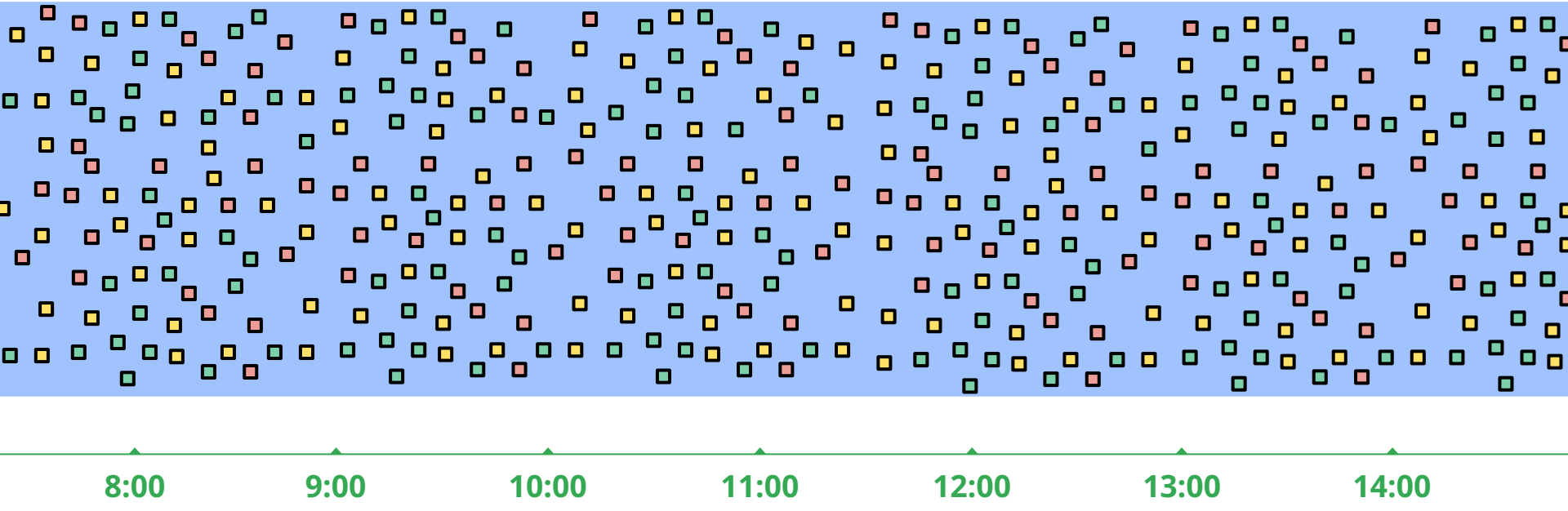
Austin, 2022

Why Streaming is Hard

We want to go from processing this:



To this:



Streaming data might be:

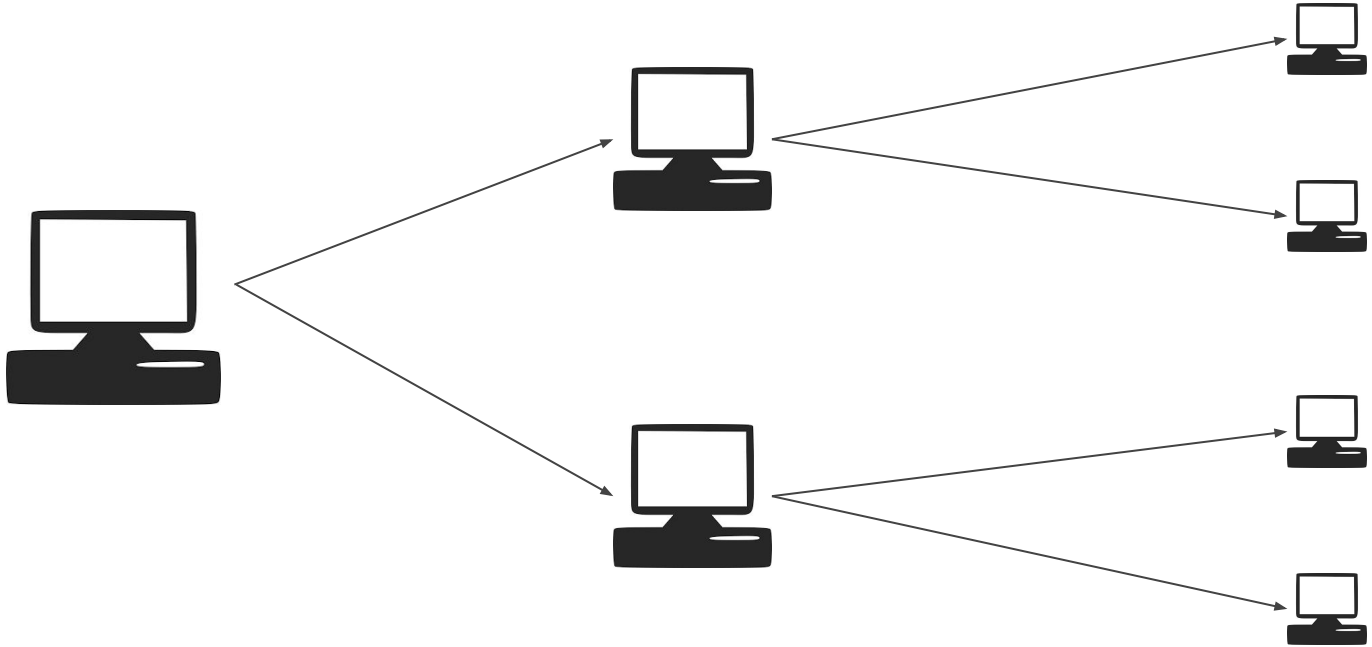
- Delayed
- Incomplete
- Rate Limited
- Infinite

You might need to:

- Aggregate over time windows
- Handle late data
- Wait for your data source to provide more data
- Update your pipeline during execution

Splittable DoFns

Scaling Up Your Processing



Motivation and Requirements

- Motivation
  - Allow the runner to scale pipeline execution to a number of workers and improve throughput.
  - Distribute long-running, parallelizable operations

- Requirements
  - Restriction Tracker - representation of data to be processed that can be split
  - Extra methods for a structural DoFn
  - Accept and use a restriction tracker in ProcessElement

Writing a Splittable DoFn

```
type splittableDoFn struct {}

func (fn *splittableDoFn) CreateInitialRestriction(filename string) offsetrange.Restriction {
        return offsetrange.Restriction {
                Start: 0,
                End: getFileLength(filename)
        }
}

func (fn *splittableDoFn) CreateTracker(rest offsetrange.Restriction) *sdf.LockRTracker {
        return sdf.NewLockRTracker(offsetrange.NewTracker(rest))
}
```

Writing a Splittable DoFn (cont.)

```
func (fn *splittableDoFn) ProcessElement(rt *sdf.LockRTracker, filename string, emit func(int)) error {
        file, err := os.Open(filename)
        if err != nil {
                return err
        }
        offset, err := seekToNextRecordBoudnaryInFile(file, rt.GetRestriction().(offsetrange.Restriction).Start)

        if err != nil {
                return err
        }
        for rt.TryClaim(offset) {
                record, newOffset := readNextRecord(file)
                emit(record)
                Offset = newOffset
        }
        return nil
}
```

Runner-Initiated Splits

- The runner can signal the worker to split its work through the Restriction Tracker into two pieces
  - Primary - work that the current worker will continue to do post-split
  - Residual - work that will be rescheduled by the runner later

- This is how work can be dynamically distributed across multiple workers

Process Continuation

What If *We* Want To Split?

- SDFs as described let the runner manage the load, but what if we have a situation where we want to split for some reason?

- In streaming workloads, we could be waiting on new data or getting throttled by the data source.

Process Continuations

- Return a Process Continuation to instruct the runner on how to handle the bundle

- Two kinds
  - Resuming - split and have the runner reschedule the remaining work after some amount of time
    - Can suggest a length of time to wait
  - Stopping - split and do not re-schedule the work, signaling that processing is done

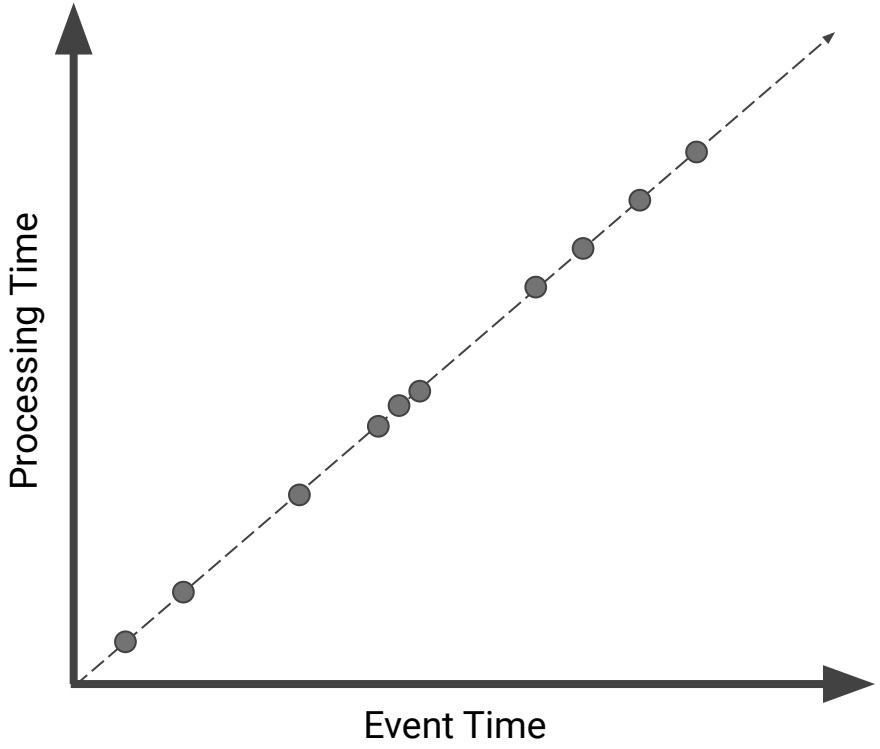
Writing a Self-Checkpointing SDF

```
func (fn *splittableDoFn) ProcessElement(rt *sdf.LockRTracker, emit func(Record)) (sdf.ProcessContinuation, error) {
        position := rt.GetRestriction().Start
        for {
                records, err := getNextRecords(position)
                if err != nil {
                        if err == ThrottlingErr {
                                return sdf.ResumeProcessingIn(60 * time.Second), nil
                        }
                        return sdf.StopProcessing(), err
                }
                for _, record := range records {
                        if !rt.TryClaim(position) {
                                return sdf.StopProcessing(), nil
                        }
                        position += 1
                        emit(record)
                }
        }
}
```

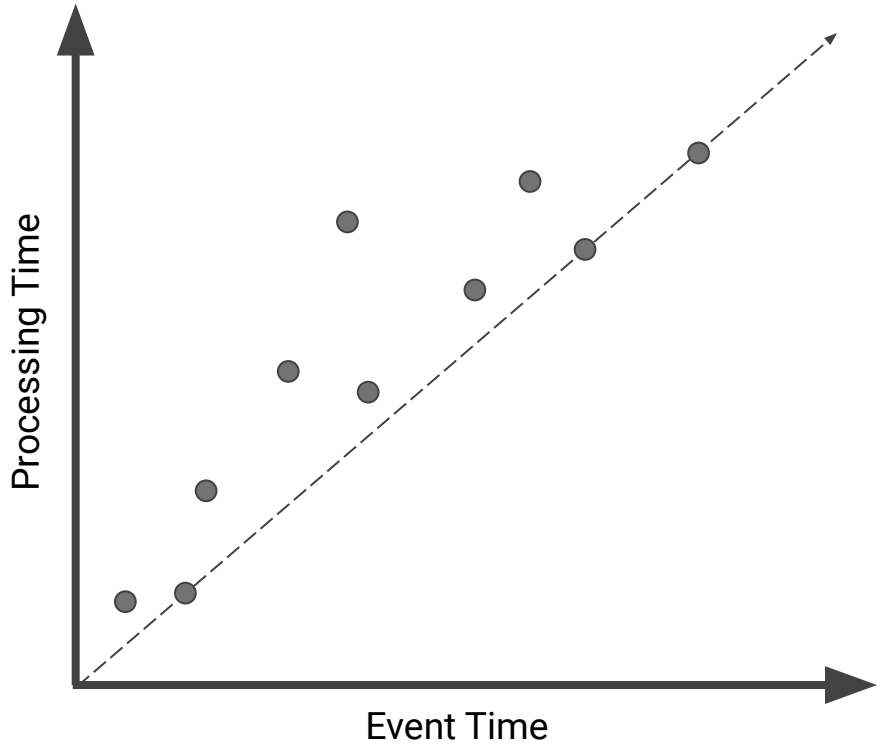
Watermark Estimation

BEAM SUMMIT Austin, 2022

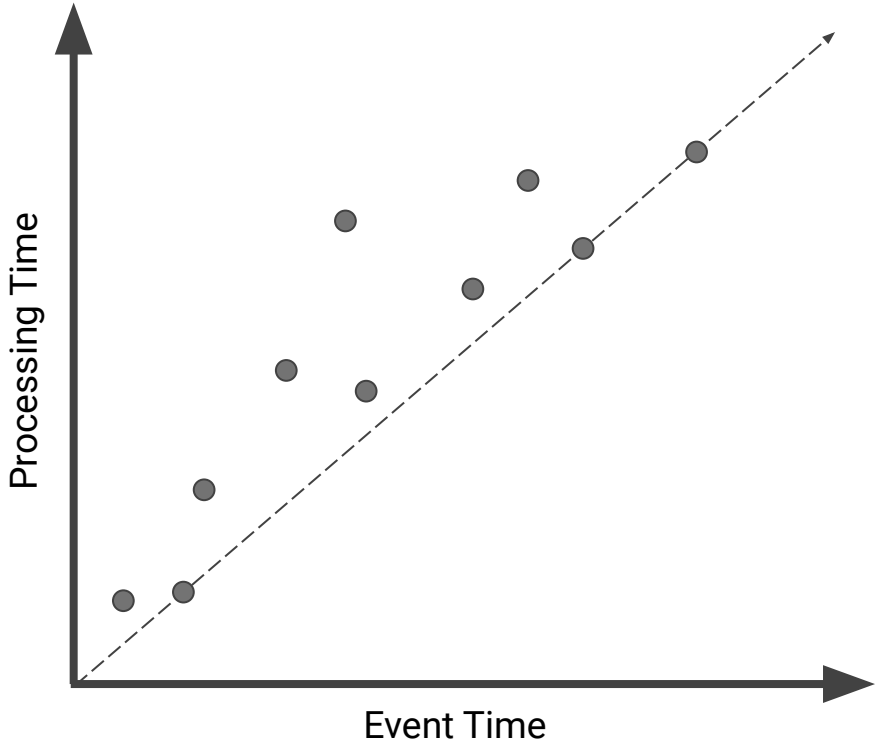
Real Time vs Event Time - Expectation



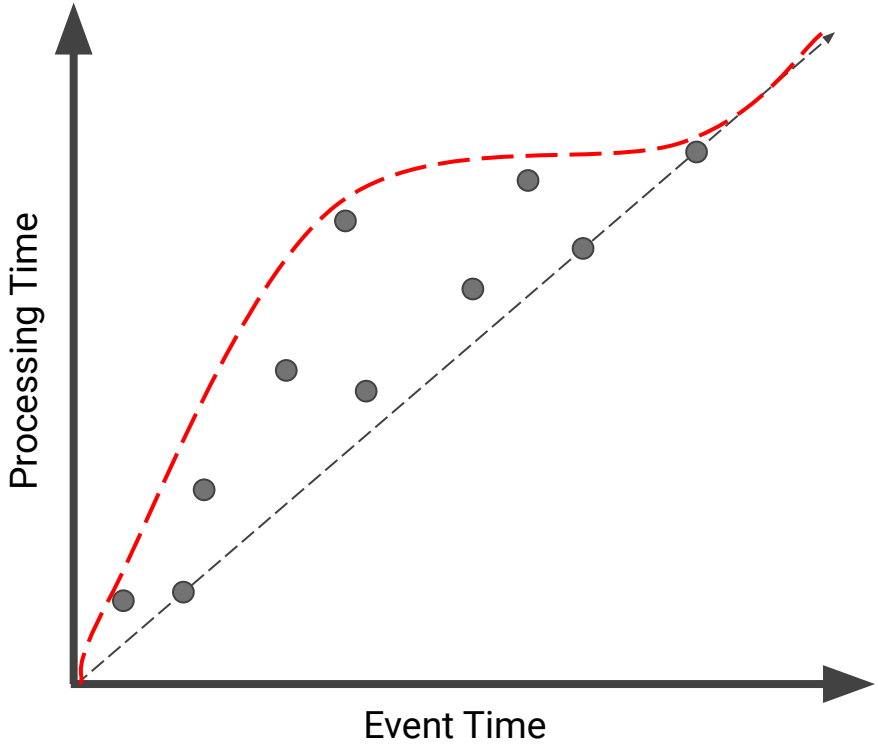
Real Time vs Event Time - Reality



How do we know its safe to finish a window's work?



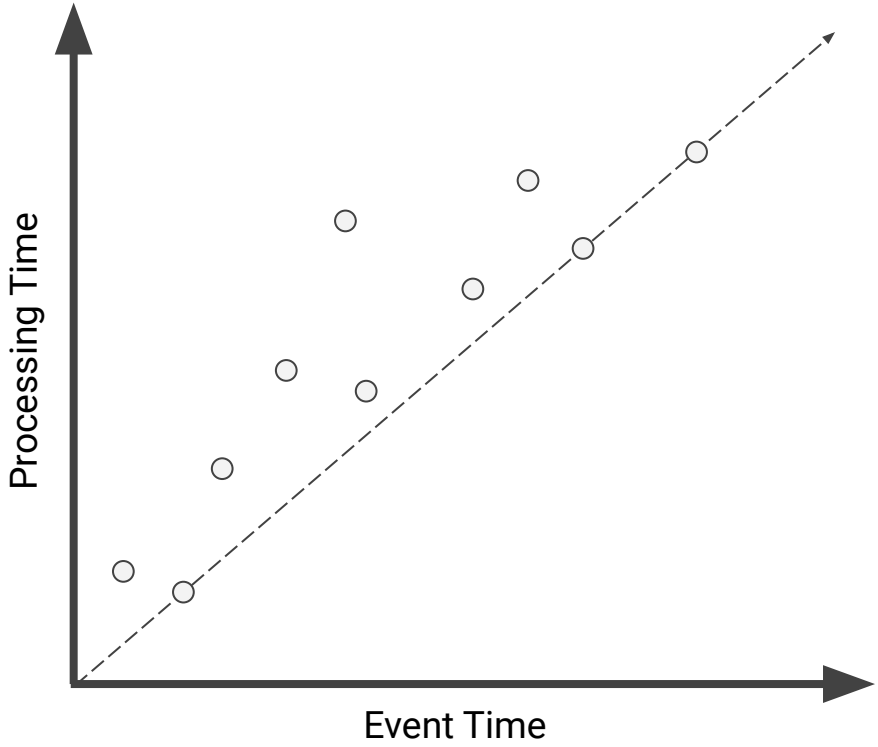
How do we know its safe to finish a window's work? Watermarks!



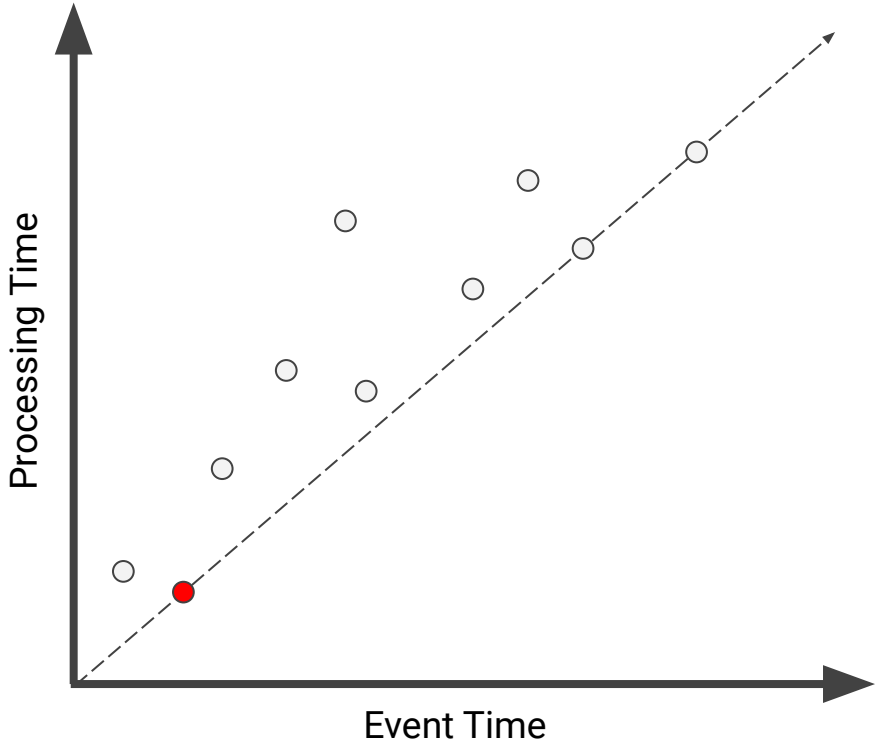
Watermarks

- Beam's notion of when data is complete
- Once a watermark passes the end of a window, additional data for that window is considered late
- Beam Go has several built in watermark estimators

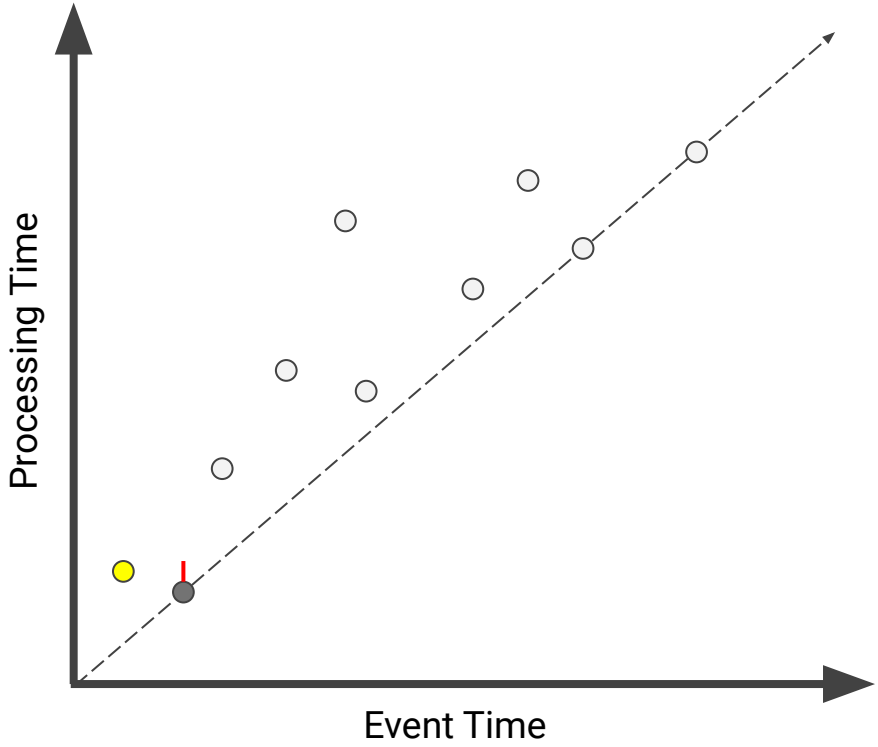
Example 1: Timestamp Observing Watermark Estimation



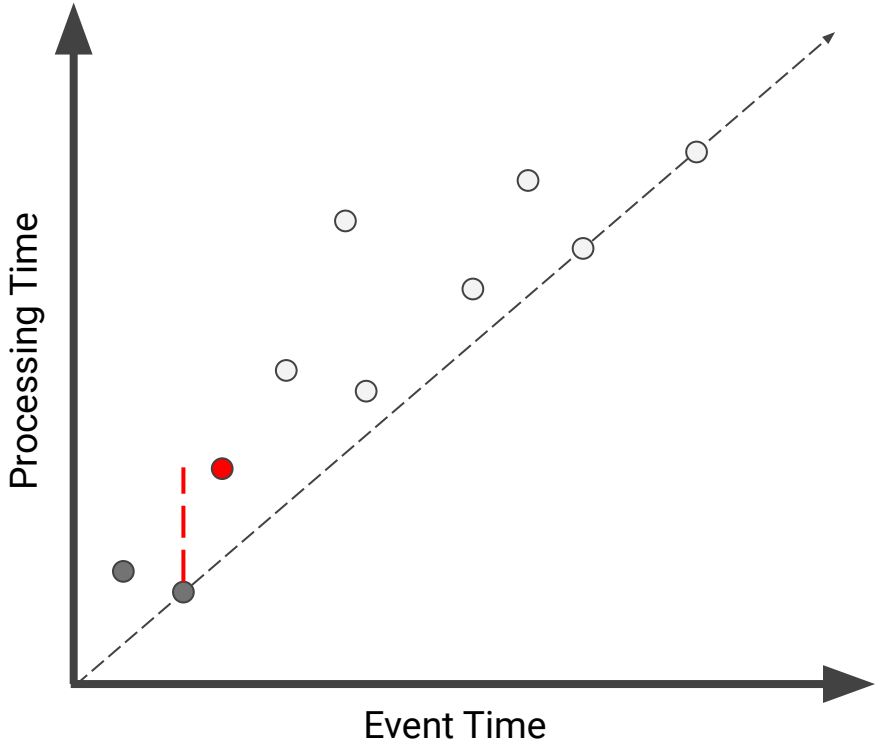
Example 1: Timestamp Observing Watermark Estimation



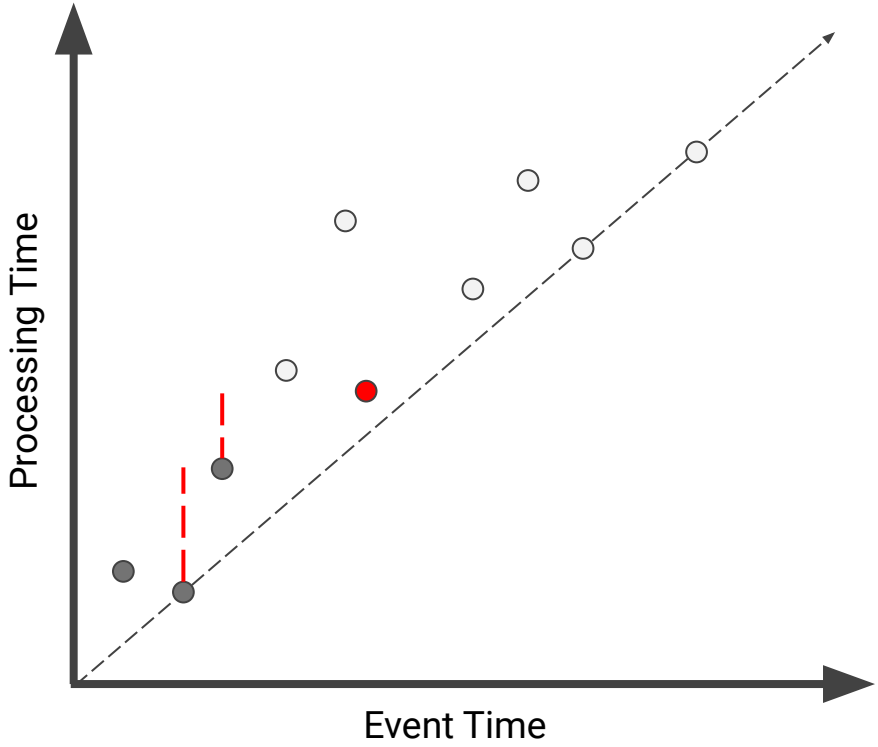
Example 1: Timestamp Observing Watermark Estimation



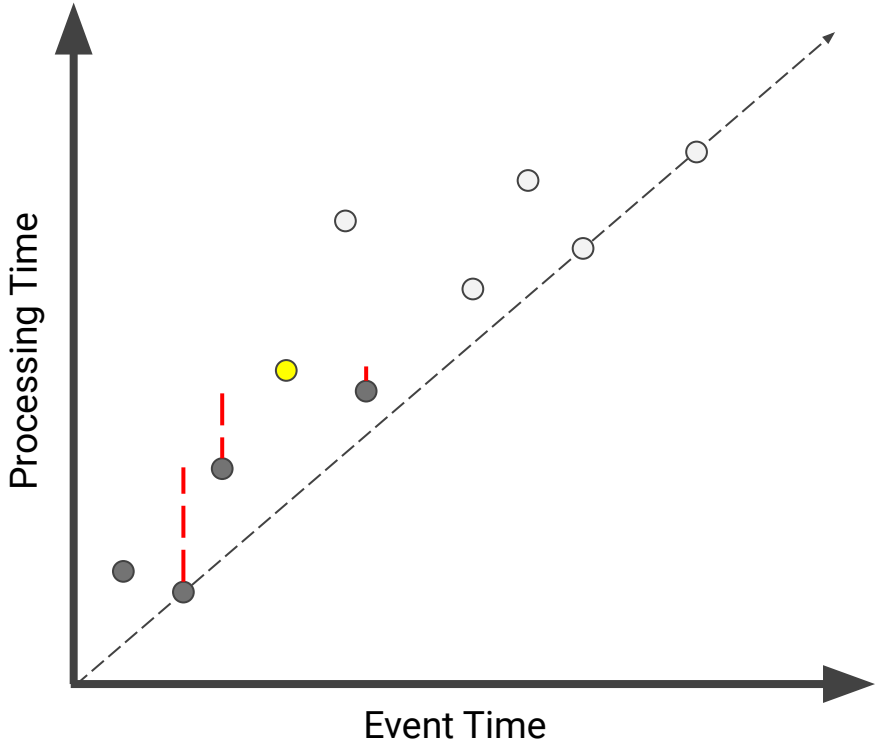
Example 1: Timestamp Observing Watermark Estimation



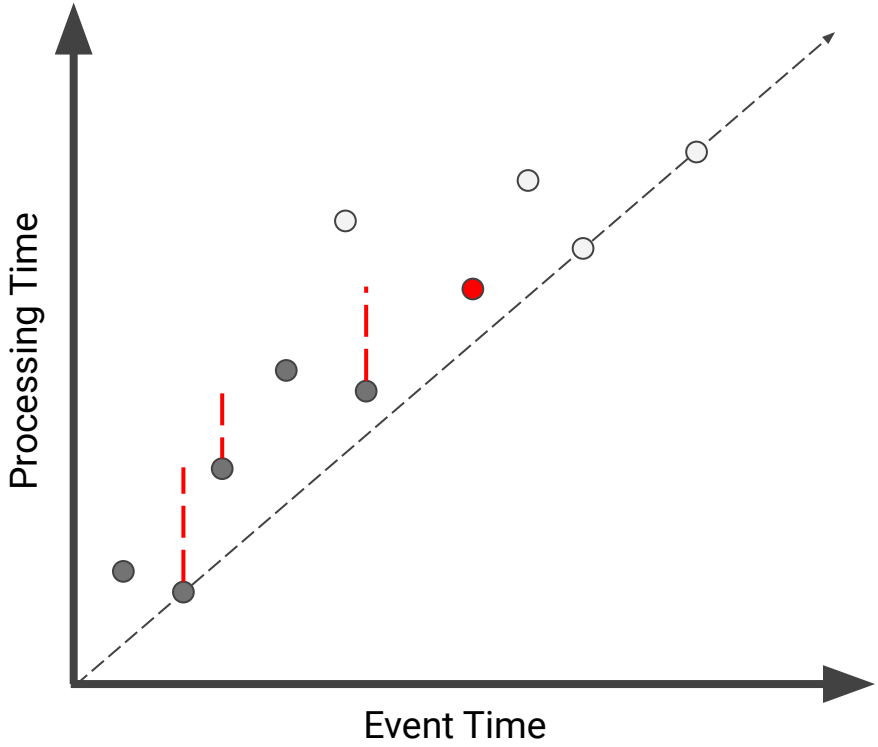
Example 1: Timestamp Observing Watermark Estimation



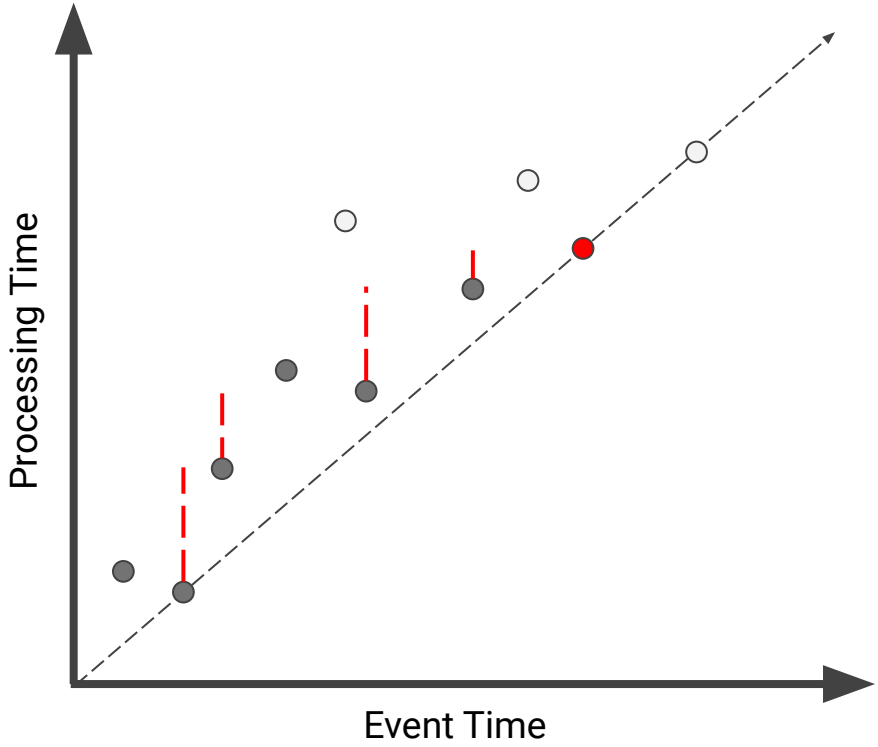
Example 1: Timestamp Observing Watermark Estimation



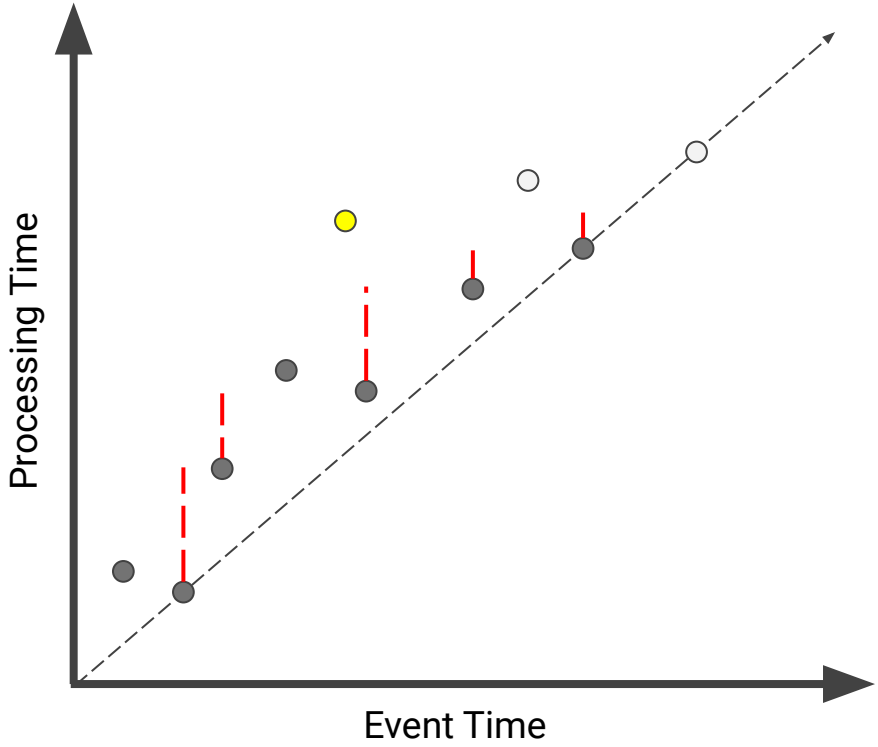
Example 1: Timestamp Observing Watermark Estimation



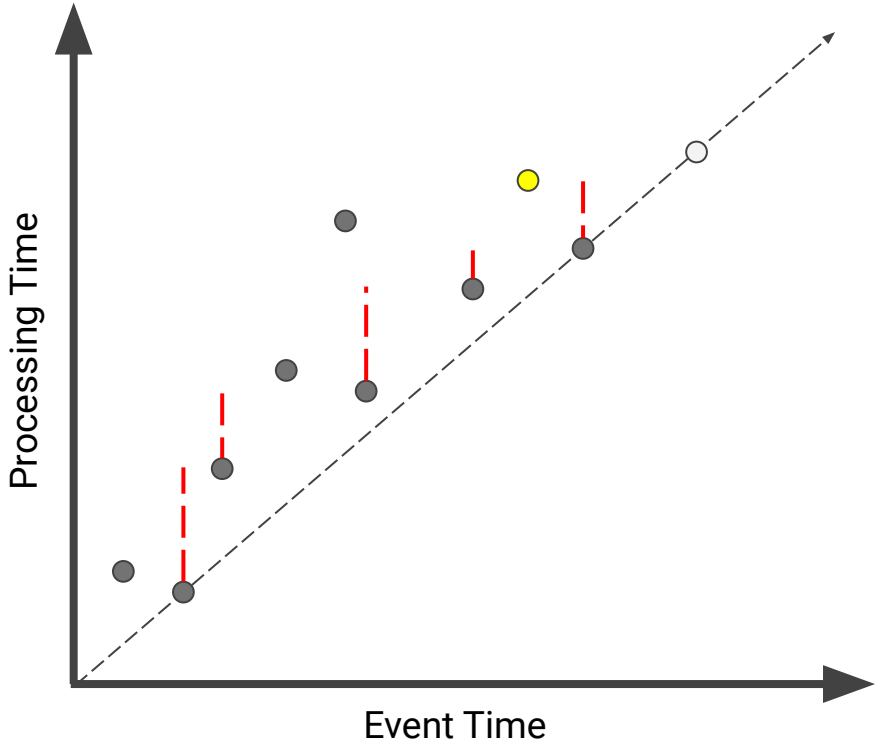
Example 1: Timestamp Observing Watermark Estimation



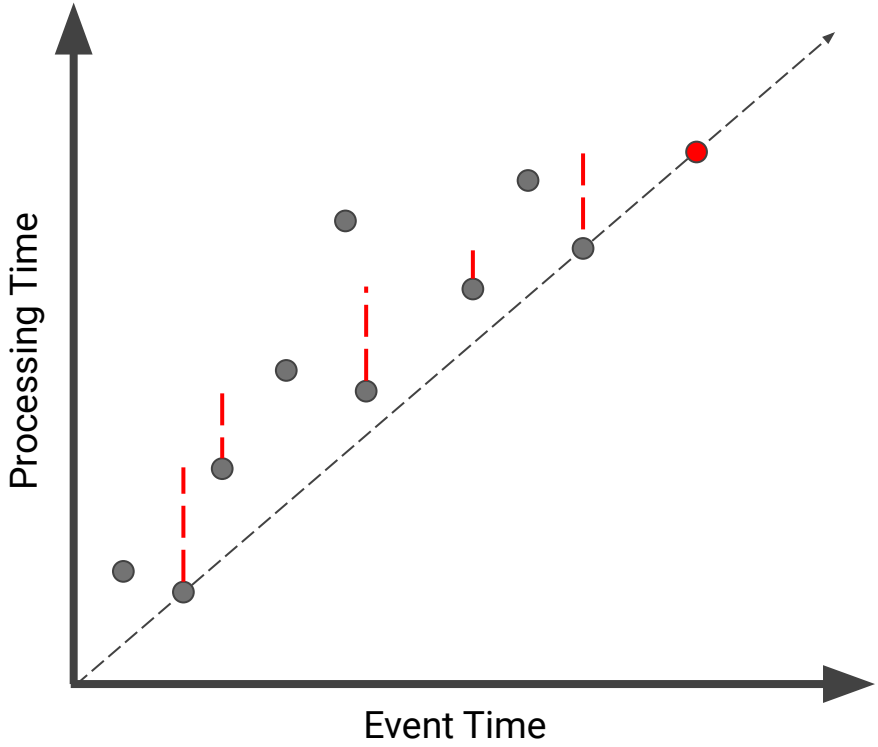
Example 1: Timestamp Observing Watermark Estimation



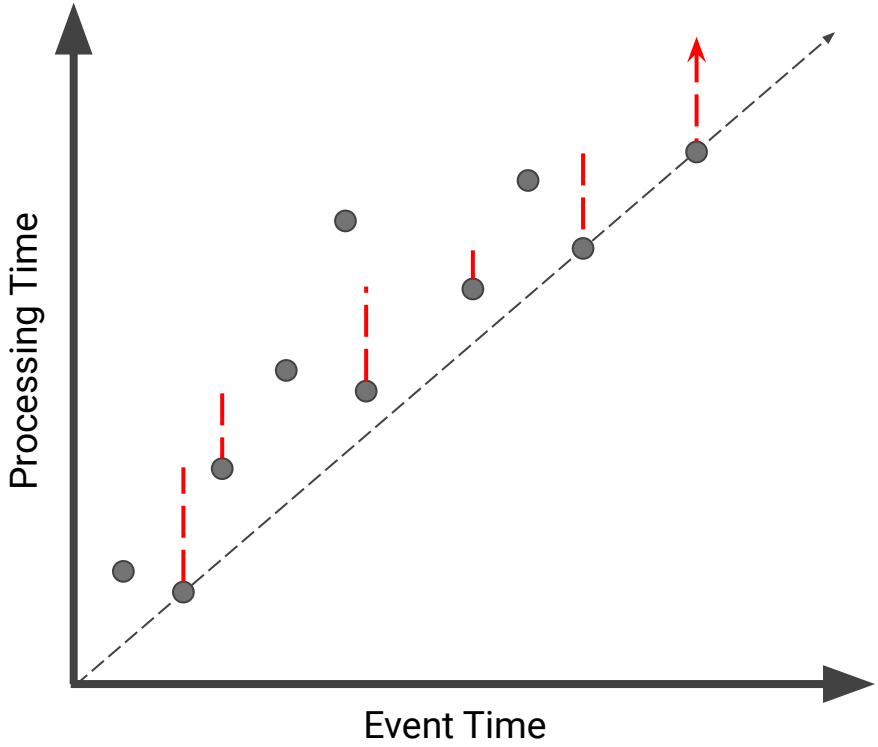
Example 1: Timestamp Observing Watermark Estimation



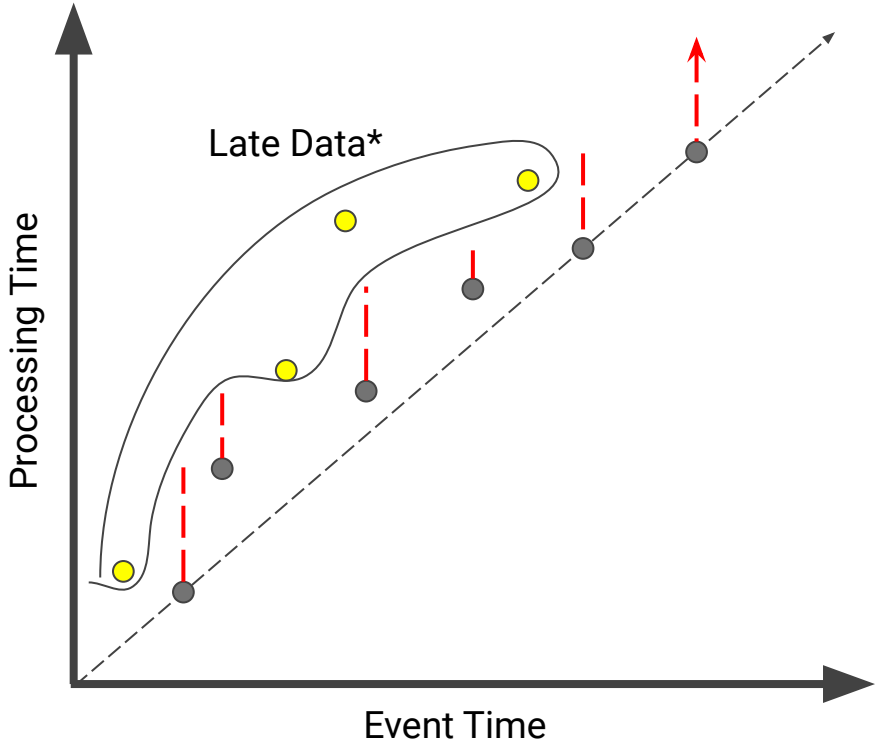
Example 1: Timestamp Observing Watermark Estimation



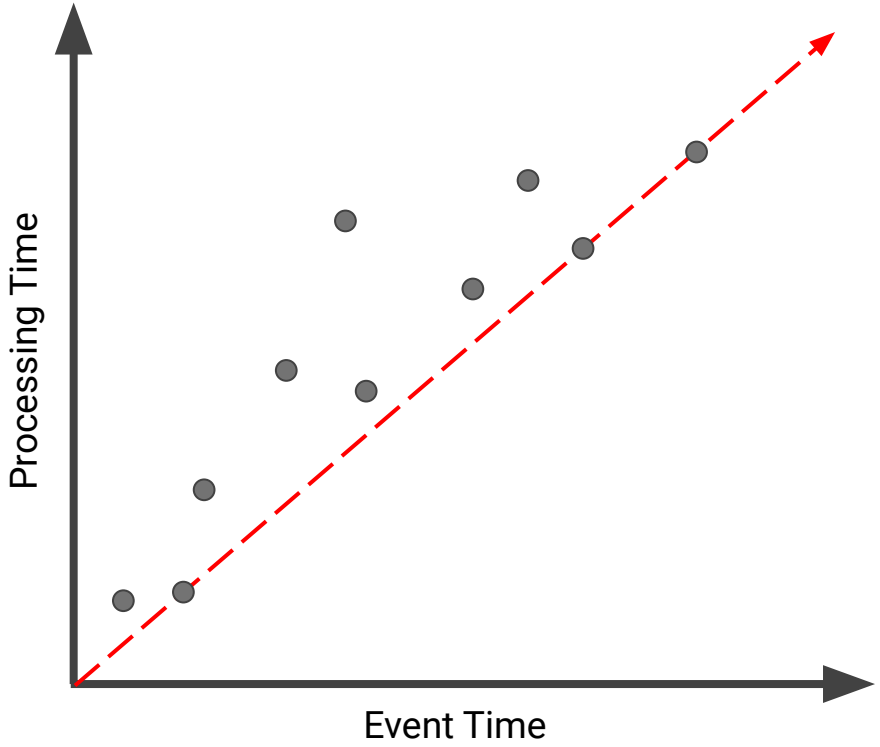
Example 1: Timestamp Observing Watermark Estimation



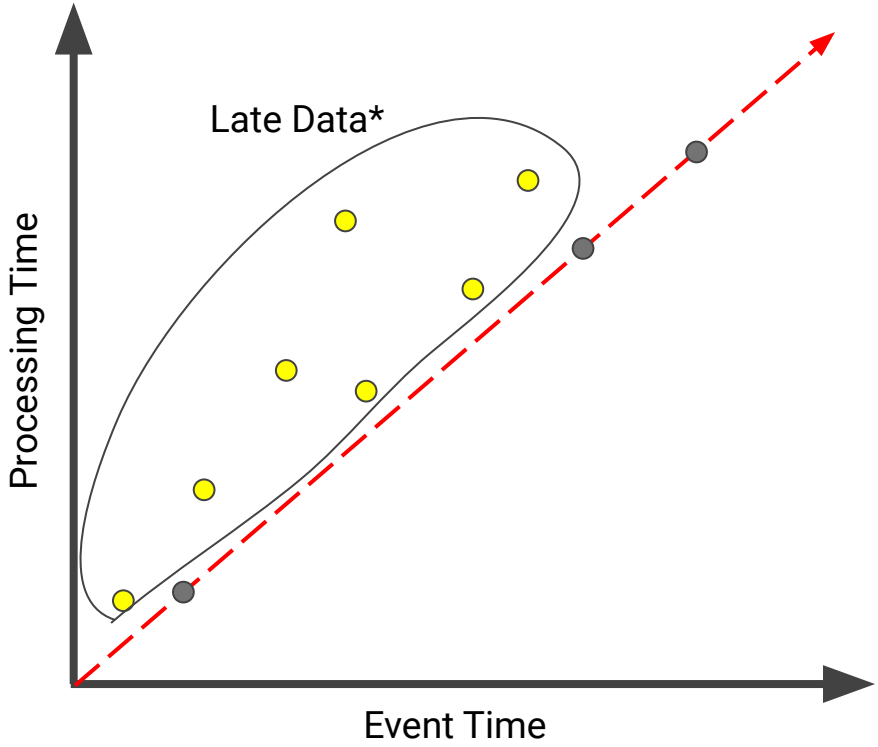
Example 1: Timestamp Observing Watermark Estimation



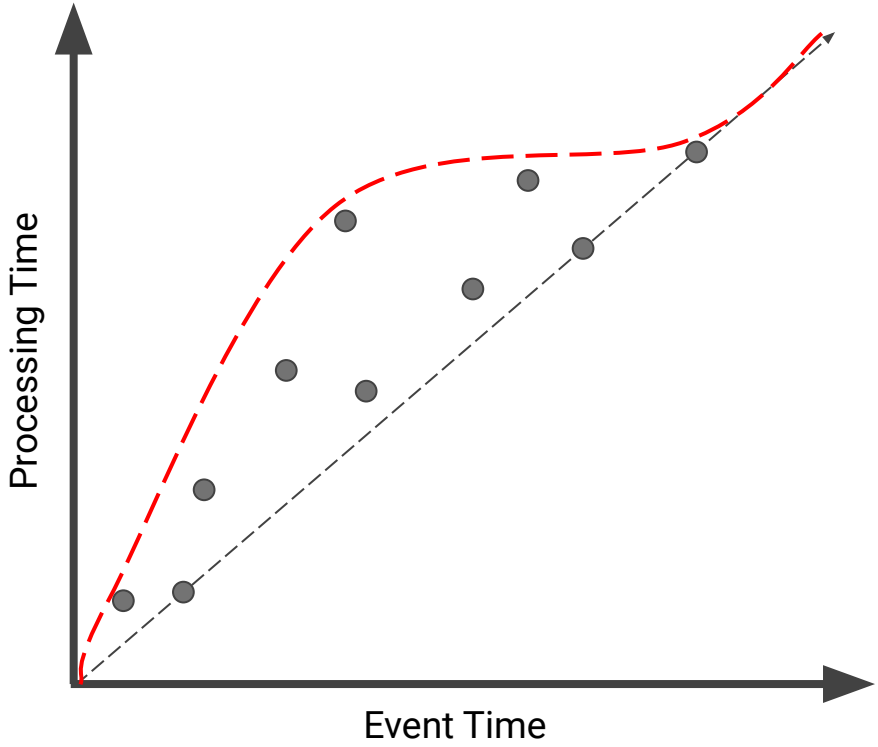
Example 2: Real Time Watermark Estimation



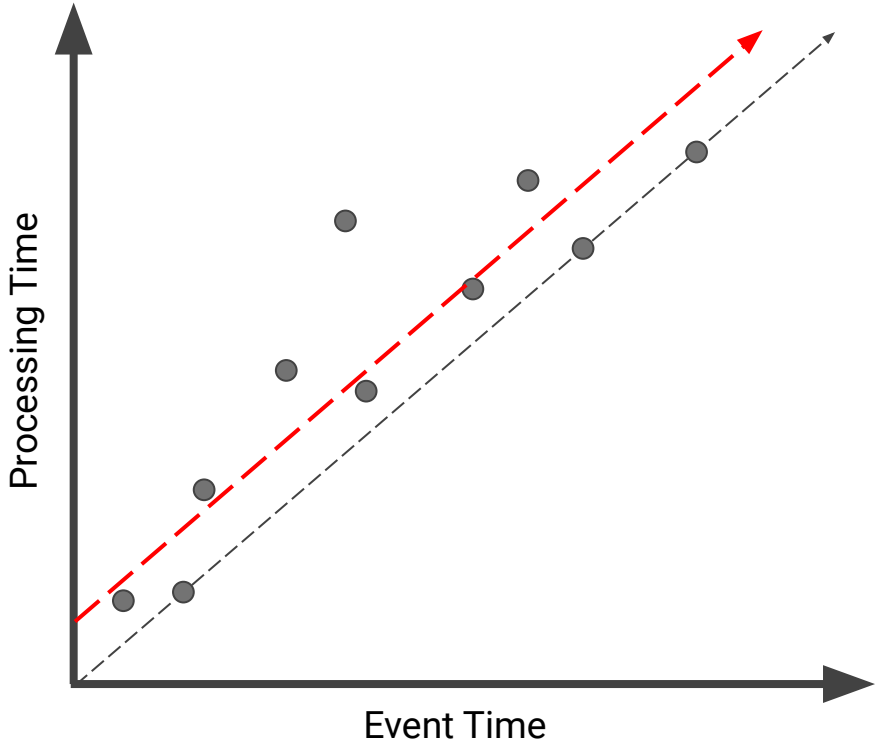
Example 2: Real Time Watermark Estimation



Example 3: Manual Watermark Estimation (Choose Your Adventure)



Example 3: Manual Watermark Estimation (Choose Your Adventure)



Creating a Custom Watermark Estimator

```
type CustomWatermarkEstimator struct {
        state WatermarkState
}

func (e *CustomWatermarkEstimator) CurrentWatermark() time.Time {
        return e.state.Watermark
}

// Optional
func (e *CustomWatermarkEstimator) ObserveTimestamp(ts time.Time) {
        e.state.Watermark = ts
}
```

Using a Watermark Estimator

```
func (fn *weDoFn) CreateWatermarkEstimator(initialState WatermarkState) *CustomWatermarkEstimator {
        return &CustomWatermarkEstimator{state: initialState}
}

func (fn *weDoFn) InitialWatermarkEstimatorState(et beam.EventTime, rest offsetrange.Restriction, element string)
WatermarkState {
        return WatermarkState{Watermark: time.Now()}
}

func (fn *weDoFn) WatermarkEstimatorState(e *CustomWatermarkEstimator) WatermarkState {
        return e.state
}

func (fn *weDoFn) ProcessElement(e *CustomWatermarkEstimator, element string) {
        // ...
        e.state.Watermark = time.Now()
}
```

Bundle Finalization

Bundle Finalization

- Register callbacks executed once runner has durably persisted output of a bundle
- Useful for acking messages
- Best effort, doesn't guarantee success or handle errors

Using Bundle Finalization

```
func (fn *splittableDoFn) ProcessElement(bf beam.BundleFinalization, rt *sdf.LockRTracker, element string) {
        // ...
        bf.RegisterCallback(5*time.Minute, func() error {
                // ... perform a side effect ...

                return nil
        })
}
```

Putting it All Together

BEAM SUMMIT Austin, 2022

Example: Native PubSub I/O

Go has cross-language PubSub streaming support, but what would a native version look like?

Restriction Tracker

```
type SubscriptionRTracker struct {
        Subscription  string
        Done          bool
}

func (fn *SubscriptionRTracker) TryClaim(pos interface{}) bool {
        posString, ok := pos.(string)
        return ok && posString == s.Subscription
}

func (fn *SubscriptionRTracker) TrySplit(frac float64) (primary, residual interface{}, err error) {
        If frac ==0.0 {
                resid := s.Subscription
                s.Subscription = “”
                s.Done = true
                return “”, resid, nil
        }
        return s.Subscription, “”, nil
}
```

Processing Elements

```
func (fn *pubSubRead) ProcessElement(ctx context.Context, bf beam.BundleFinalization, rt *sdf.LockRTracker, _ []byte,
                                     emit func(beam.EventTime, []byte)) (sdf.ProcessContinuation, error) {
        // …
        timeout := time.NewTimer(5*time.Second)
        for {
                select {
                case m, ok := <- messChan:
                        r.processedMessages = append(r.processedMessages, m)
                        emit(beam.EventTime(m.PublishTime.UnixMilli()), m.Data)
                        timeout.Reset(5*time.Second)
                case <- timeout.C:
                        return sdf.ResumeProcessingIn(10*time.Second), nil
                }
        }
}
```

Bundle Finalization

```
func (fn *pubSubRead) ProcessElement(ctx context.Context, bf beam.BundleFinalization, rt *sdf.LockRTracker, _ []byte,
                                     emit func(beam.EventTime, []byte)) (sdf.ProcessContinuation, error) {
        bf.RegisterCallback(5*time.Minute, func() error {
                for _, m := range r.processedMessages {
                        m.Ack()
                }
                r.processedMessages = nil
                return nil
        })
        // …
}
```

Questions?

GitHub - damccorm, jrmccluskey
Twitter - @jrmccluskey

BEAM SUMMIT Austin, 2022