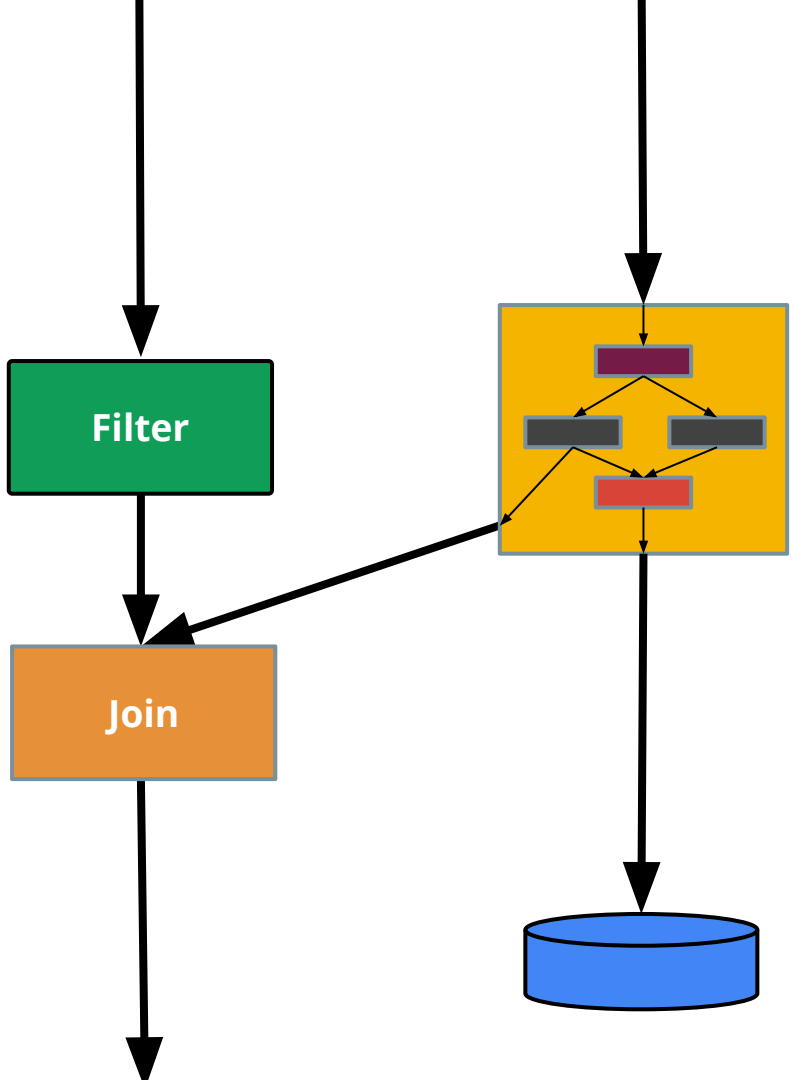
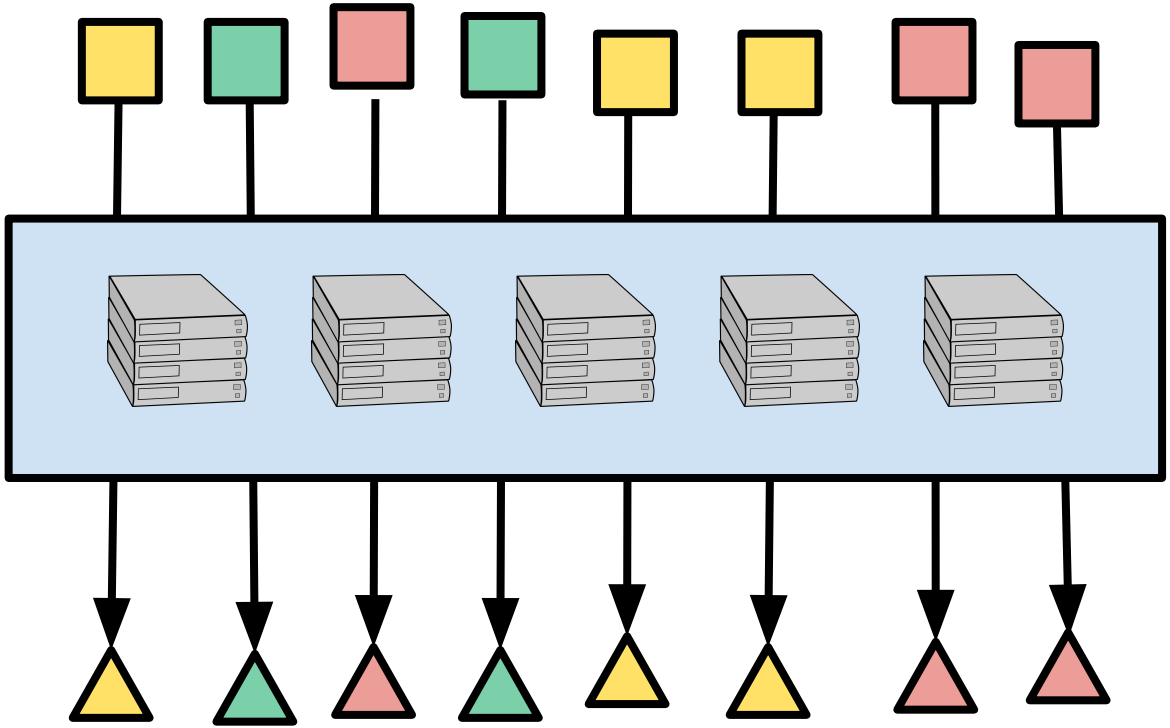
State & Timers Design Patterns

Reza Rokni
Kenn Knowles


Filter
Join

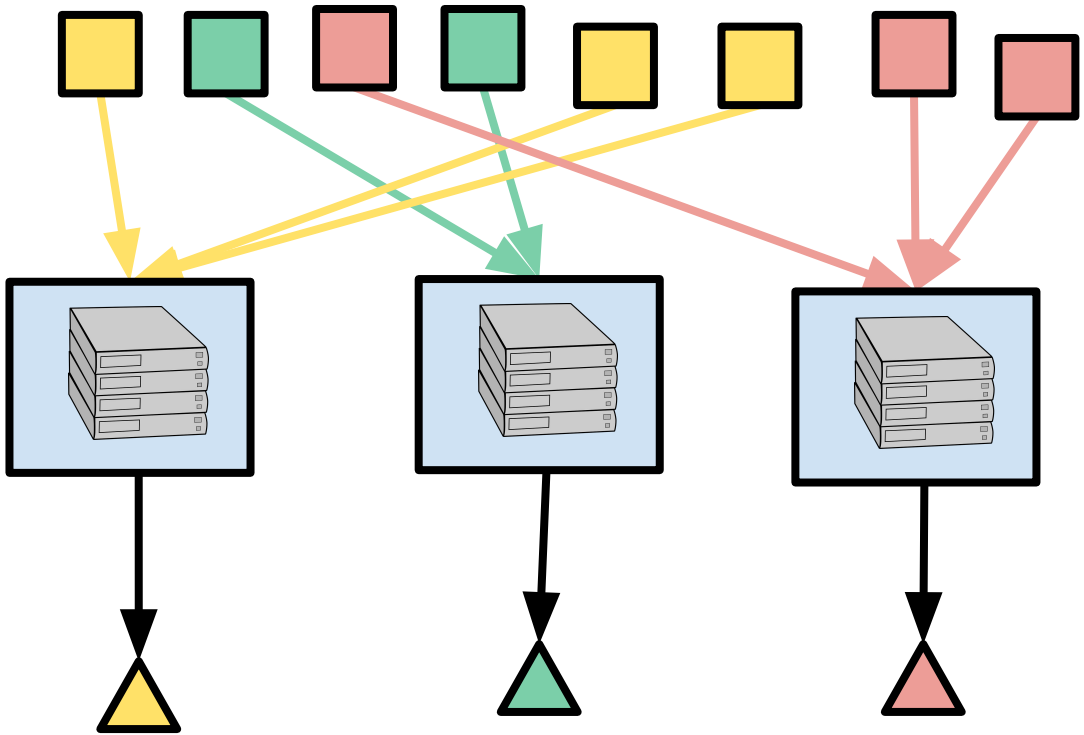
Per element: `ParDo` (`Map`, etc)



Every item processed independently

Stateless implementation

Per key: `Combine` (`Reduce`, etc)



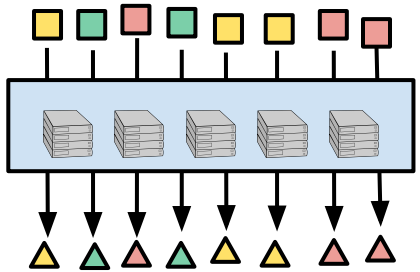
Items grouped by some key and combined

Stateful streaming implementation (buffering until trigger)

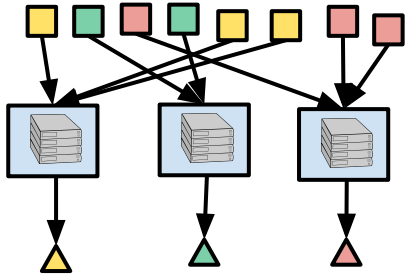
But your code doesn't work with state, just associative & commutative function

It "just works" with massive out-of-order streams

ParDo, Map, etc.



Combine, Reduce, etc.



"Parse incoming events and filter out bad data"

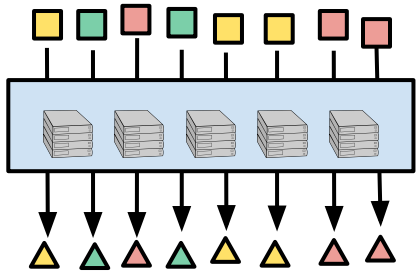
"Sum per hour and output when you have the whole hour"

"Put events in 10 minute windows sliding every 2 minutes"

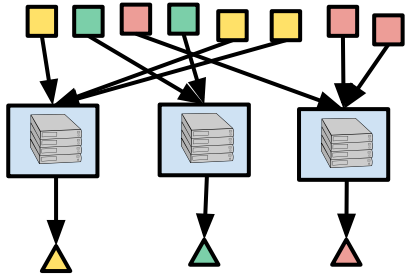
"Group into sessions and emit as fast as possible"

But what if you need more control?

ParDo, Map, etc.



Combine, Reduce, etc.



"I'm getting tied in knots trying to do this with windows & triggers"

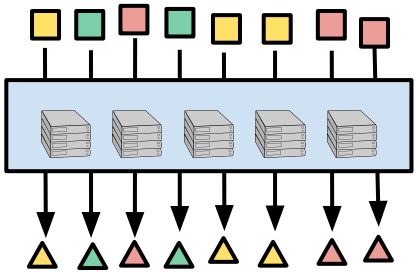
"My aggregation is not an associative & commutative operator"

"Triggers aren't specific enough for my use case"

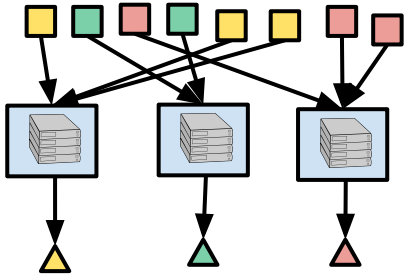
"I need to output even when data isn't coming in"

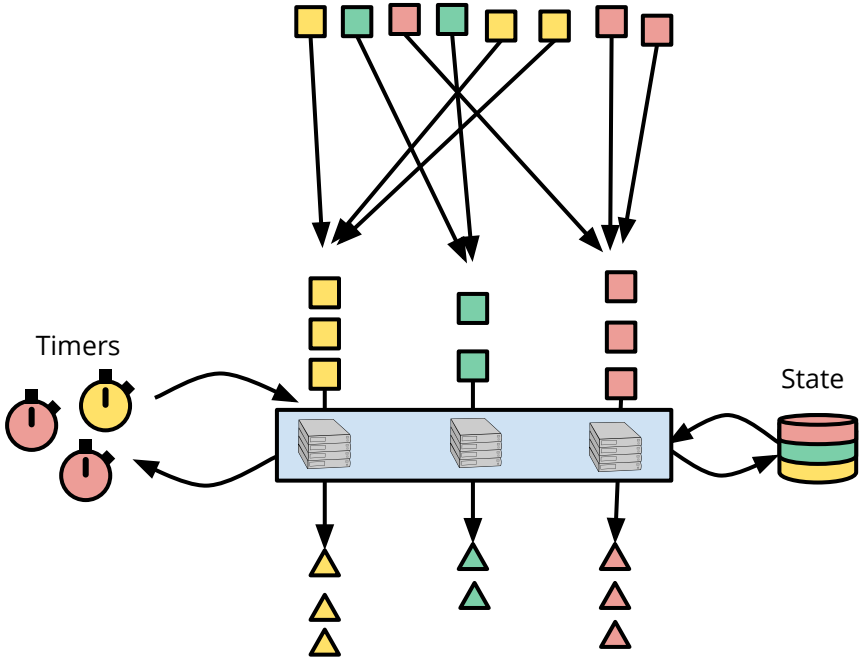
Beam primitive: state & timers

ParDo, Map, etc.



Combine, Reduce, etc.





What it looks like (in Java)

```
new DoFn<KV<String, Foo>, Baz>() {

  @StateId("counter")
  private final StateSpec<Object, ValueState<Integer>> indexSpec = StateSpecs.value();

  @ProcessElement
  public void processElement(@StateId("counter") ValueState<Integer> counter, ...) {
    int current = firstNonNull(counter.read(), 0);
    counter.write(current+1);
    ...
  }
}
```

What it looks like (in Java)

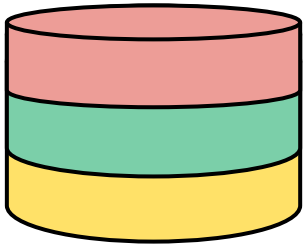
```
new DoFn<KV<String, Foo>, Baz> {

  @TimerId("flush-timer")
  private TimerSpec flushSpec = TimerSpecs.timer(TimeDomain.PROCESSING_TIME);

  @ProcessElement
  public void process(@TimerId("flush-timer") Timer flushTimer, …) {
    flushTimer.setForNowPlus(…); …
  }

  @OnTimer("flushTimer")
  public void onTimer(…) {
      … // read state, output values
  }
}
```

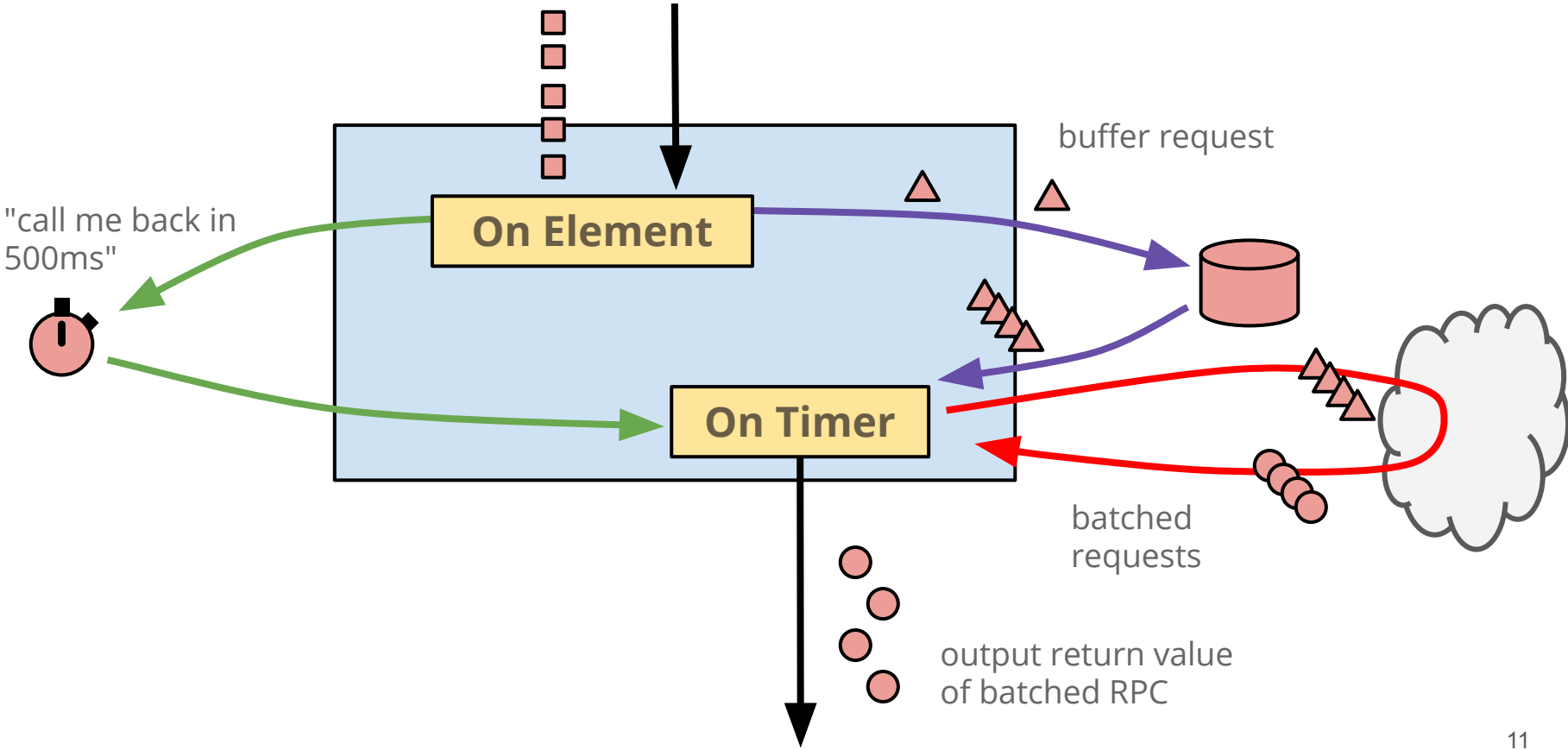
State is per key and window



Key	Window	MEDIAN_IDLE	MAIN_ACTIVITY	...
"kenn"	9am - 10am	10m	"hack"	
	12pm - 1pm	25m	"eat"	
	11pm - 12am	60m	"sleep"	
"reza"	8am - 9am	20m	"flight"	
	11am - 12pm	3m	"hack"	
...	...			

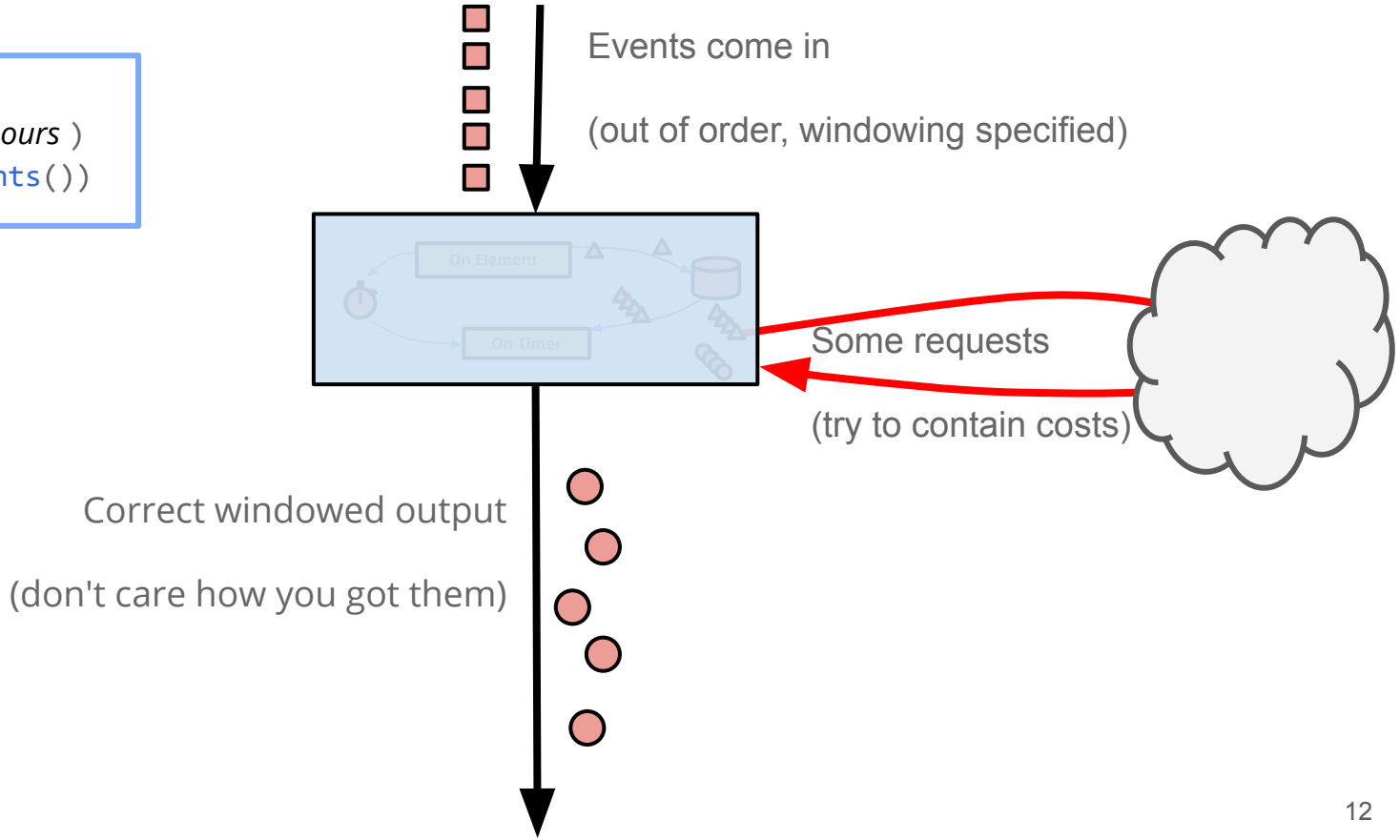
Bonus: automatically garbage collected when a window expires
(vs manual clearing of per-key state)

Example: time-batched requests

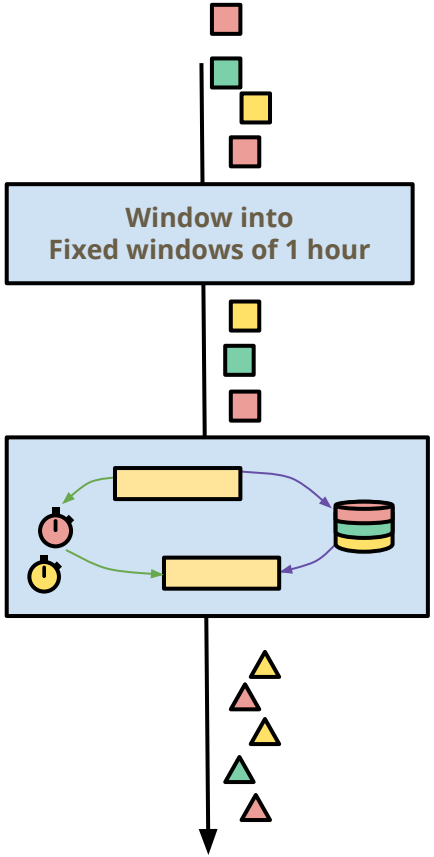


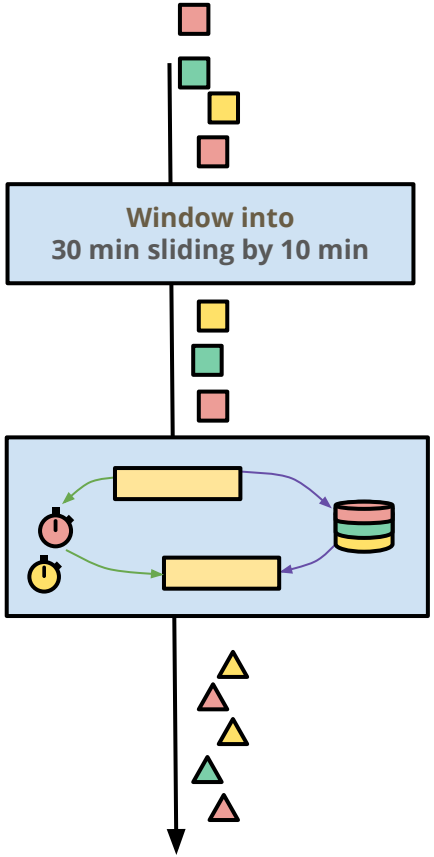
User's view of your transform

```
input
  .apply(Window.into( hours )
  .apply(new EnrichEvents())
```

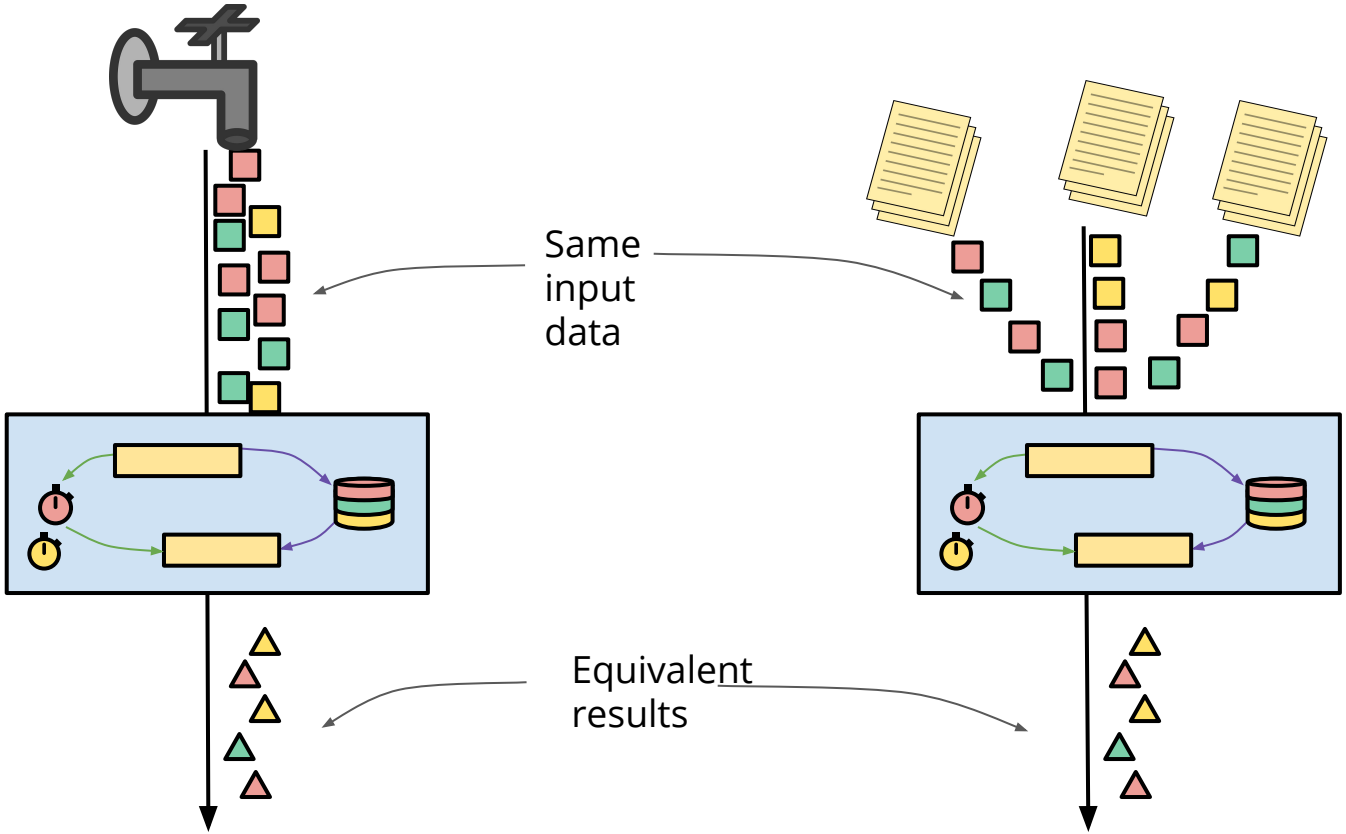


Event time windowing still "just works"





Unified present & historical processing



Types of state

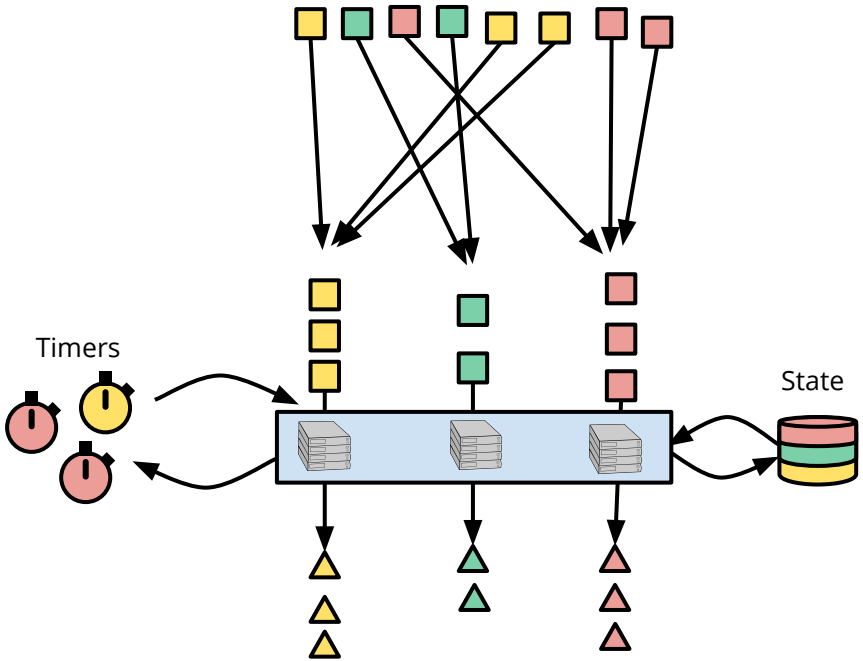
Type	Strength	Dataflow Runner
`Value`	Read/write any value (but always the whole value)	Yes
`Bag`	Cheap append No ordering on read	Yes
`Combining`	Associative/commutative compaction	Yes
	Membership checking	No
`Map`	Read/write just keys you specify	No

Not yet supported in Dataflow runner

Types of timers

Type	Uses
`Processing time`	Timeouts Relative times ("in 5 minutes") Periodically output based on state
`Event time`	Output based on completeness of input data Absolute times ("when the data is complete up to 5:00am") Final/authoritative outputs Don't leave data behind in state!

Isn't this just "the only" primitive? (no)



You: express more complex logic

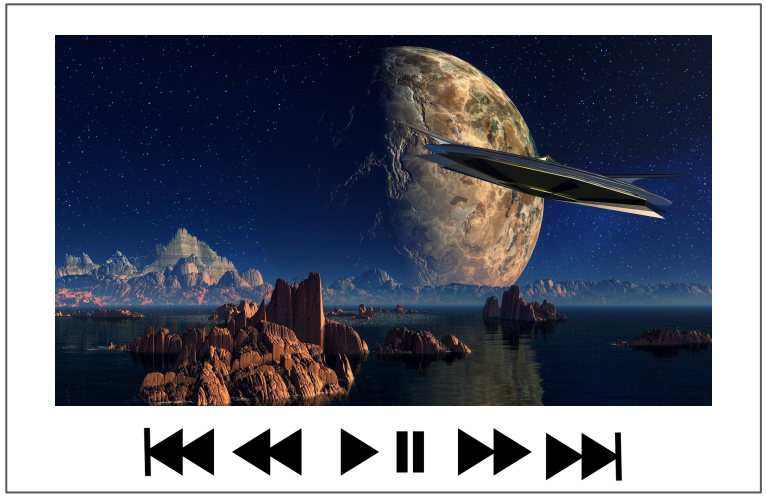
Runner: fewer optimization/execution choices

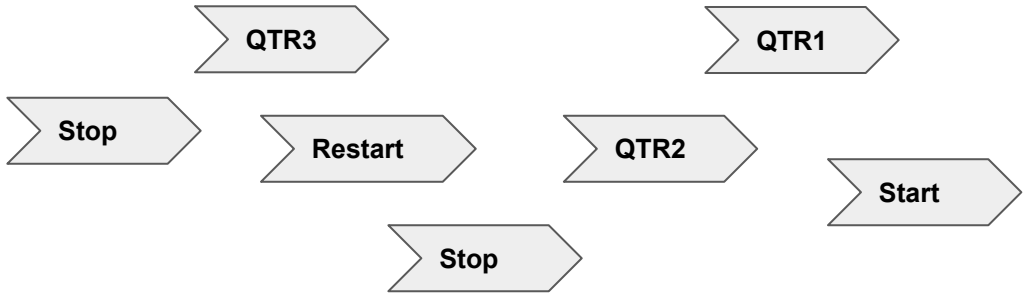
Most of your pipeline should still be the simpler stuff

What else can you do with state & timers

- **Domain-specific triggering** ("output when five people who live in Seattle have checked in")
- **Slowly changing dimensions** ("update FX rates for currency ABC")
- **Stream joins** ("join-matrix" / "join-biclique")
- **Fine-grained aggregation** ("add odd elements to accumulator A and event elements to accumulator B")
- **Per-key workflows** (like user sign up flow w/ reminders & expiration)

Walk through of a non-trivial example!





Events coming from player devices

https://pixabay.com/vectors/music-play-pause-icons-next-back-1956799/
https://pixabay.com/photos/house-home-quirky-movie-hobbit-2616607/

Warning boilerplate ahead ... look away now...

State and Timers is a raw set of API's

Dependent on use case, lots of boilerplate may be needed;

- Order is not guaranteed..
- The Timer API does not support read() operation
- OnTimer() does not have key context
- Order is not guaranteed..
- DoFn.StartBundle and DoFn.FinishBundle can not access State
- Did we mention Order is not guaranteed?

Laying out the rules...

Implement some rules around events coming from our video application:

The application has the following Events:

```
@DefaultCoder(SchemaCoder.class)
public enum UserEventType {
 START(0),
 QTR_1(1),
 QTR_2(2),
 QTR_3(3),
 END(4),
 PAUSE(5),
 RESTART(6),
 STOP(7);
}
```

Laying out the rules...

Given those events we would like to implement these rules

Rule 0 Output and GC When we have END or no activity for 10 Mins no allowed lateness

Rule 1 Output every time we see a QTR_3

Rule 2 Output when we see STOP but no other activity for 2 Min's

Rule 3 Output when we see multiple RESTART's without any QTR updates during GC

Rule 0 - Setup State Objects

Rule 0 Output and GC When we have END or no activity for 10 Mins (no allowed lateness)

```
@TimerId("ttl")
private final TimerSpec ttl = TimerSpecs.timer(TimeDomain.EVENT_TIME);

@StateId("key")
private final StateSpec<ValueState<String>> key = StateSpecs.value();

@StateId("maximumTimestampObserved")
private final StateSpec<ValueState<Instant>> maximumTimestampObserved = StateSpecs.value();
```

Rule 0 - Setup State Objects

Rule 0 Output and GC When we have END or no activity for 10 Mins no allowed lateness

```
@ProcessElement
public void processElement(
   @Element KV<String, UserEventType> event,
   @TimerId("ttl") Timer ttl,
   @StateId("key") ValueState<String> key,
@StateId("maximumTimestampObserved") ValueState<Instant> maximumTimestampObserved,
...
```

Rule 0 - Process Events

Rule 0 Output and GC When we have END or no activity for 10 Mins no allowed lateness

```
@ProcessElement
...
// Current limitation of OnTimer()is it does not store the key
key.write(event.getKey());

// Set last value processed Do NOT EXPECT ORDERED INPUTS!
if (maximumTimestampObserved.read() == null) {
 lastProcessedEvent.write(timestamp);
} else {
 if (maximumTimestampObserved.read().isBefore(timestamp)) {
   maximumTimestampObserved.write(timestamp);
 }
}
// Rule 0 GC after 10 min's of no activity.. no allowed lateness
ttl.set(maximumTimestampObserved.read().plus(Duration.standardMinutes(10)));
```

Rule 0 - Garbage Collect

Rule 0 Output and GC When we have END or no activity for 10 Mins no allowed lateness

```
@OnTimer("ttl")
public void ttl(
   @StateId("key") ValueState<String> key,
   @StateId("maximumTimestampObserved") ValueState<Instant> maximumTimestampObserved,
....
   OutputReceiver<KV<String, SystemEventType>> o) {
....

o.output(KV.of(key.read(), SystemEventType.GC));

maximumTimestampObserved.clear();
key.clear();
...
```

Rule 1 - Simple filter

Rule 1 Output every time we see a QTR_3

```
@ProcessElement
...

// Check for Rule 1 : Output when we are at QTR_3
if (event.getValue().equals(UserEventType.QTR_3)) {
 o.output(KV.of(event.getKey(), SystemEventType.RULE_1));
}
```

Rule 2 - Setup State Objects

Rule 2 Output when we see STOP but no other activity for 2 Min's

```
@StateId("key")
private final StateSpec<ValueState<String>> key = StateSpecs.value();

@StateId("maximumTimestampObserved")
private final StateSpec<ValueState<Instant>> maximumTimestampObserved = StateSpecs.value();

@TimerId("rule2")
private final TimerSpec rule2 = TimerSpecs.timer(TimeDomain.EVENT_TIME);
```

Rule 2 - Catch Stop & Set Timer

Rule 2 Output when we see STOP but no other activity for 2 Min's

Note: Timer object does not have reset / delete option

```
// Check for Rule 2 Output when we see STOP but no RESTART for 2 Min's
if (event.getValue().equals(UserEventType.STOP)) {
  rule2.set(timestamp.plus(Duration.standardMinutes(2)));
}
```

Rule 2 - Process Stop

Rule 2 Output when we see STOP but no other activity for 2 Min's

```
   @OnTimer("rule2")
   public void rule2(
       @StateId("key") ValueState<String> key,
       @StateId("maximumTimestampObserved") ValueState<Instant> maximumTimestampObserved,
       OnTimerContext otc) {
     // If no new events since STOP timer was set output event
     // Assume if an event came in with time == stop time, STOP is later
     if (otc.timestamp().isAfter(maximumTimestampObserved.read())
         && !otc.timestamp().isEqual(maximumTimestampObserved.read()))
       otc.output(KV.of(key.read(), SystemEventType.RULE_2));
   }
 }
}
```

Rule 3 - Setup State Objects

Rule 3 Output when we see multiple RESTART's without any QTR updates, at Rule 0

```
@StateId("key")
private final StateSpec<ValueState<String>> key = StateSpecs.value();

@StateId("events")
private final StateSpec<BagState<TimestampedValue<Integer>>> events = StateSpecs.bag();

@StateId("maximumTimestampObserved")
private final StateSpec<ValueState<Instant>> maximumTimestampObserved = StateSpecs.value();
```

Rule 3 - Process Restart

Rule 3 Output when we see multiple RESTART's without any QTR updates, at Rule 0

We can use Rule 0 GC function rather than setup a new OnTimer()

```
// Check for Rule 3 : Output when we see RESTART more than once in session without a QTR
// update

if (event.getValue().equals(UserEventType.RESTART)) {
  eventsBag.add(TimestampedValue.of(1, timestamp));
}
```

Rule 3 - Test in GC

Rule 3 Output when we see multiple RESTART's without any QTR updates at Rule 0

```
@OnTimer("ttl")
public void ttl(
    @StateId("key") ValueState<String> key,
    @StateId("maximumTimestampObserved") ValueState<Instant> maximumTimestampObserved,
    @StateId("events") BagState<TimestampedValue<Integer>> eventsBag,
    OutputReceiver<KV<String, SystemEventType>> o) {

  // Implement Rule 3
  Instant maxQTRTime = Instant.EPOCH;
  int restartCount = 0;
  for (TimestampedValue<Integer> events : eventsBag.read()) {
    if (events.getValue() == 0) {
      maxQTRTime =
          (maxQTRTime.isAfter(events.getTimestamp())) ? events.getTimestamp() : maxQTRTime;
    }
  }
```

Rule 3 - Test in GC

Rule 3 Output when we see multiple RESTART's without any QTR updates at Rule 0

We can use Rule 0 GC function rather than setup a new OnTimer()

```
for (TimestampedValue<Integer> events : eventsBag.read()) {
  if (events.getValue() == 0) {
    maxQTRTime =
        (maxQTRTime.isAfter(events.getTimestamp())) ? events.getTimestamp() : maxQTRTime;}}
for (TimestampedValue<Integer> events : eventsBag.read()) {
  if (events.getValue() == 1) {
    // Is this restart after max QTR time?
    restartCount += (maxQTRTime.isBefore(events.getTimestamp())) ? 1 : 0;}}

if (restartCount > 0) {
  o.output(KV.of(key.read(), SystemEventType.RULE_3));
}
// If there was two or more RESTARTS after last QTR then send out a event
o.output(KV.of(key.read(), SystemEventType.GC));
```

Summary

- Use State and Timers when you need more control
- State is per key per window
- State and Timer Types

Thank you!

Questions?

Solving for Time Series

Time Series

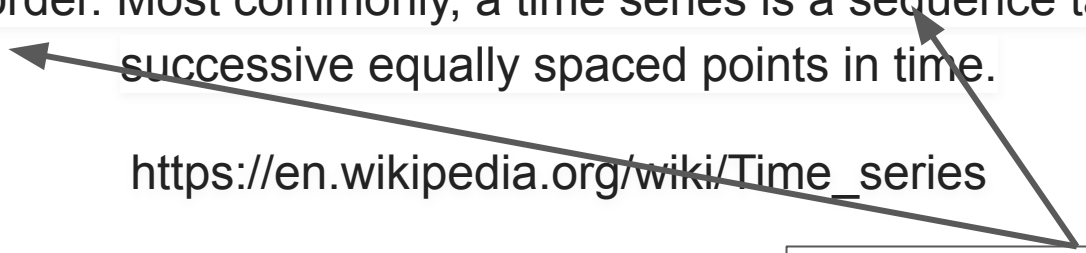
A **time series** is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.

https://en.wikipedia.org/wiki/Time_series

Timeseries

A **time series** is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.

https://en.wikipedia.org/wiki/Time_series

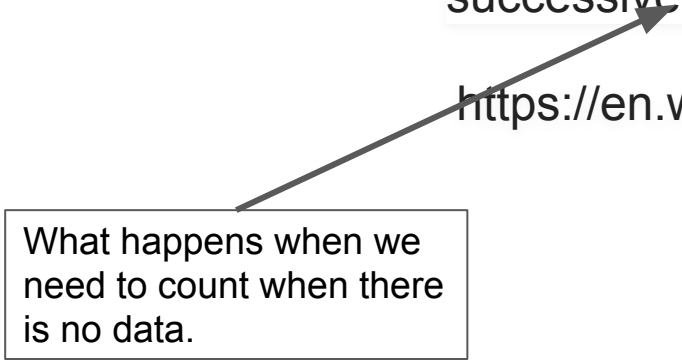


Words Order / Sequence...
Not easy to deal with ...

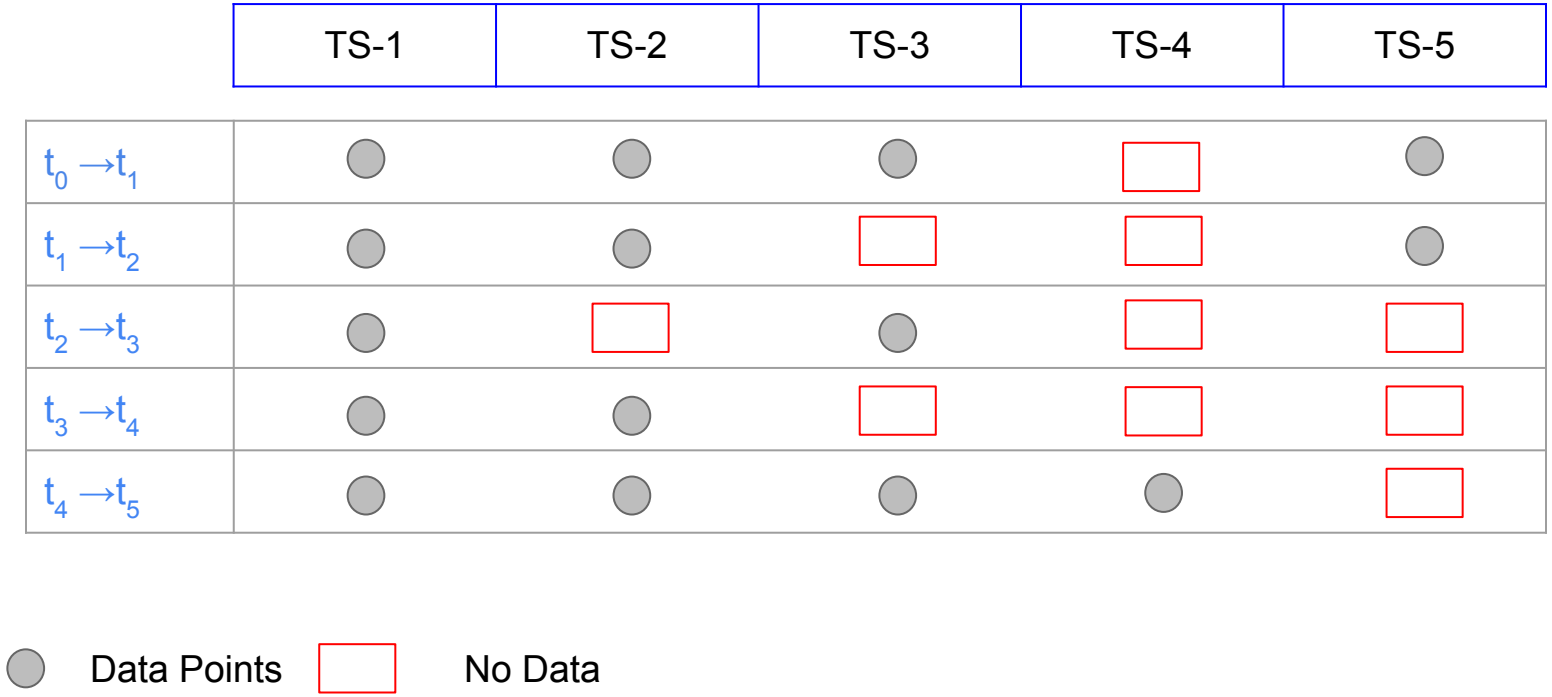
Timeseries

A **time series** is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.

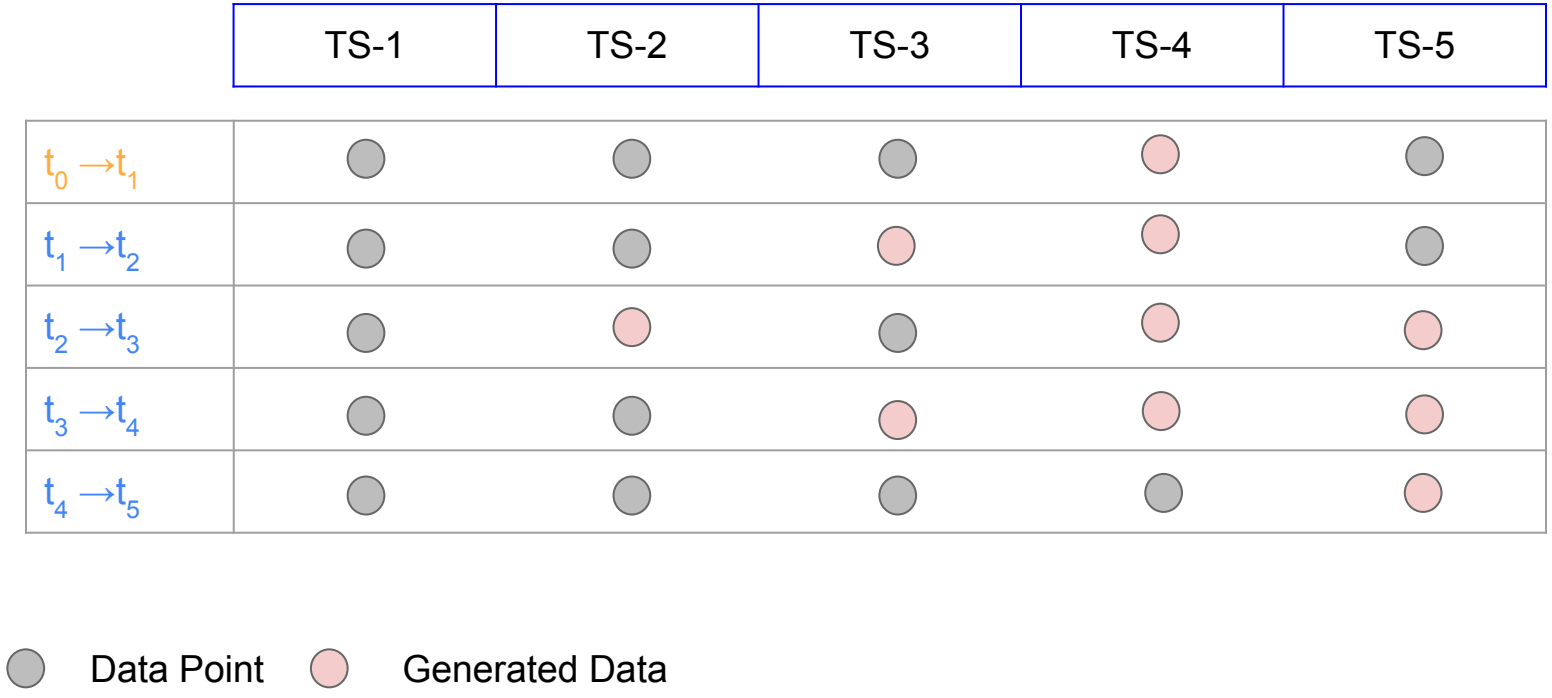
https://en.wikipedia.org/wiki/Time_series



Gap filling


TS-1
TS-2
TS-3
TS-4
TS-5
$t_0 \rightarrow t_1$
$t_1 \rightarrow t_2$
$t_2 \rightarrow t_3$
$t_3 \rightarrow t_4$
$t_4 \rightarrow t_5$
Data Points
No Data

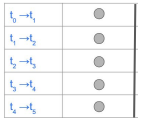
Gap filling


TS-1
TS-2
TS-3
TS-4
TS-5
$t_0 \rightarrow t_1$
$t_1 \rightarrow t_2$
$t_2 \rightarrow t_3$
$t_3 \rightarrow t_4$
$t_4 \rightarrow t_5$
Data Point
Generated Data

Options

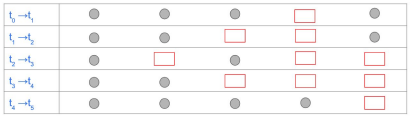
- Option 1 : Create a heartbeat message external to Pipeline
  - Bounded / UnBounded source creating an impulse at every time interval.
- Option 2 : Create a heartbeat message internal to Pipeline
  - Use of impulses from utility classes, in java Generate Sequence.
- Option 3 : Looping timers
  - Make use of self setting Timers using the Apache Beam Timer API

Option 1 External HeartBeat

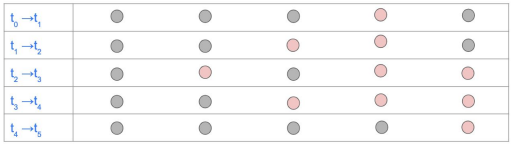


External heartbeats

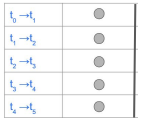
Timeseries Source



Flatten

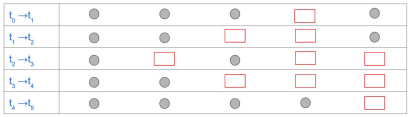


Option 1 External HeartBeat - Needs Fanout



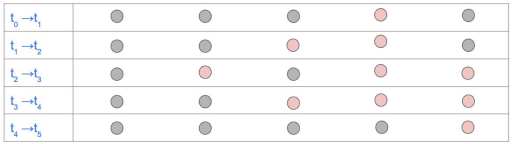
External heartbeats

Timeseries Source

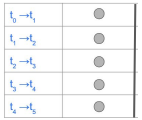


Fanout 1 -> k1,k2,k3

Flatten



Option 1 External HeartBeat - Changing keys

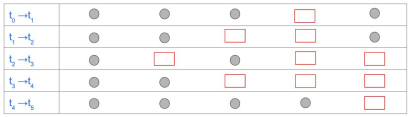


External heartbeats

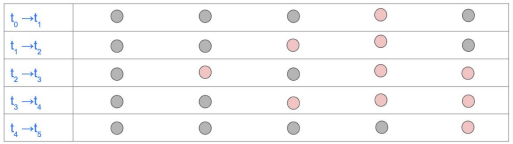
Dynamic Key List

Fanout 1 -> k1,k2,k3

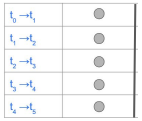
Timeseries Source



Flatten

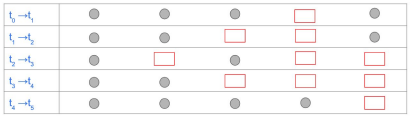


Option 1 External HeartBeat - Changing keys



External heartbeats

Timeseries Source

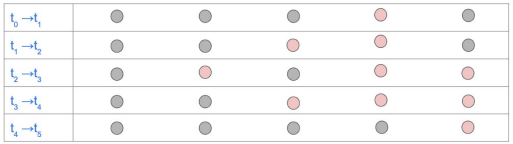


Dynamic Key List

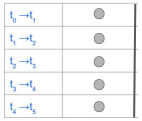
Distinct<K>

Fanout 1 -> k1,k2,k3

Flatten

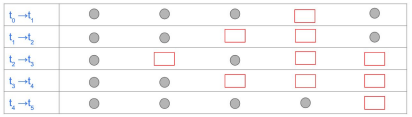


Option 2 Internal HeartBeat



GenerateSequence heartbeats

Timeseries Source

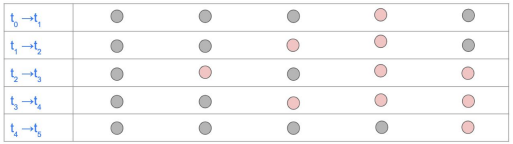


Dynamic Key List

Distinct<K>

Fanout 1 -> k1,k2,k3

Flatten



Option 3 - Looping Timer

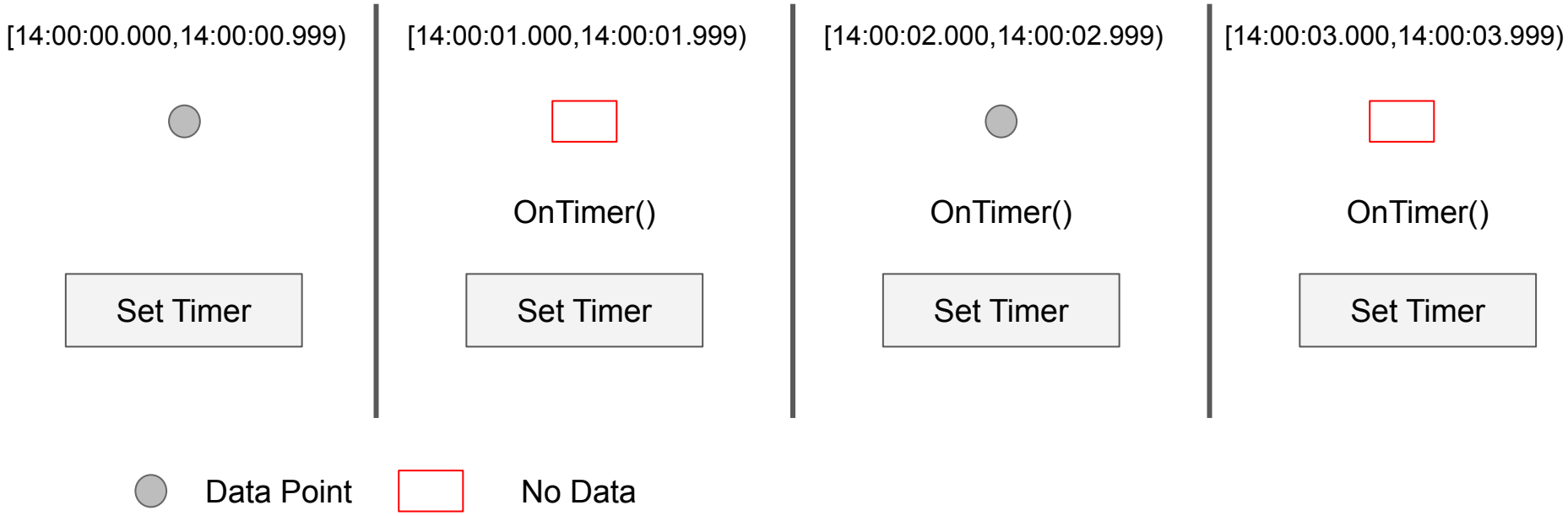
What is a Timer:

public interface Timer

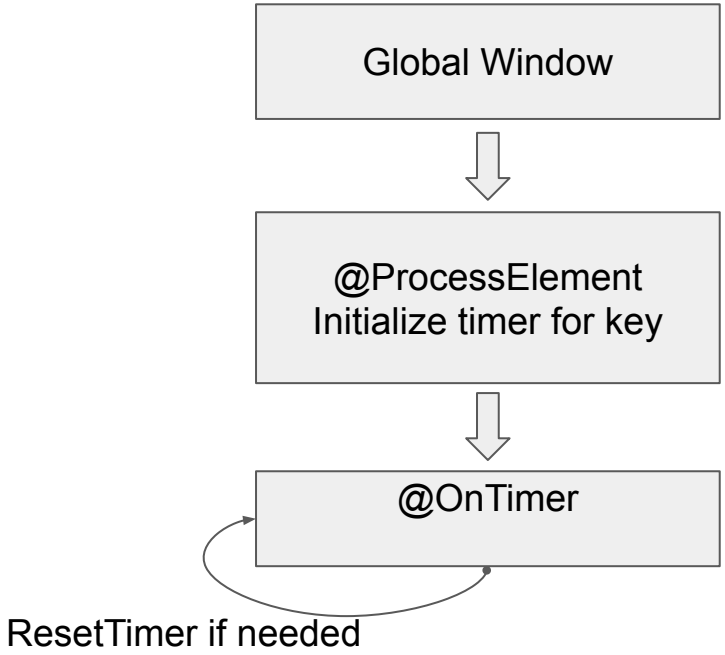
A timer for a specified time domain that can be set to register the desire for further processing at particular time in its specified time domain.

In my head... you get to set an alarm...

Looping Timers



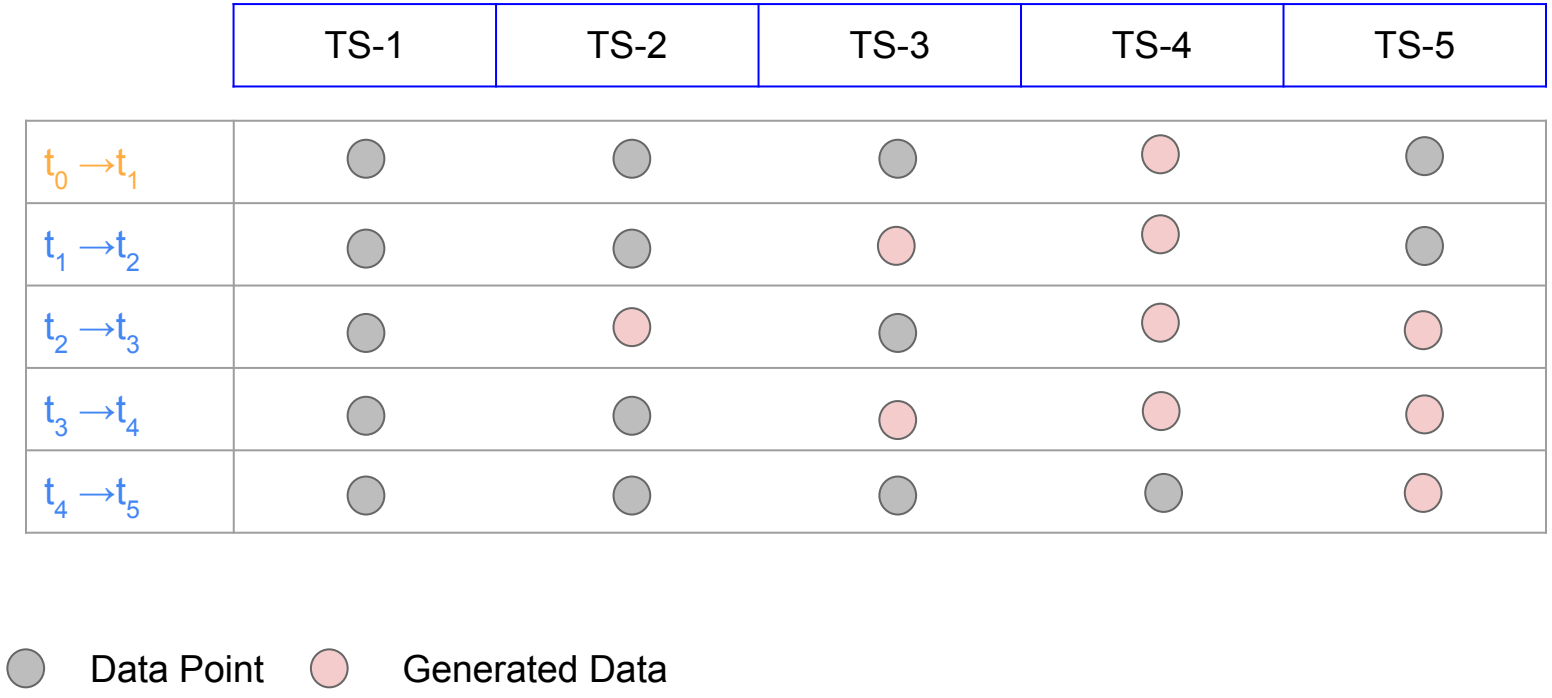
Looping Timer



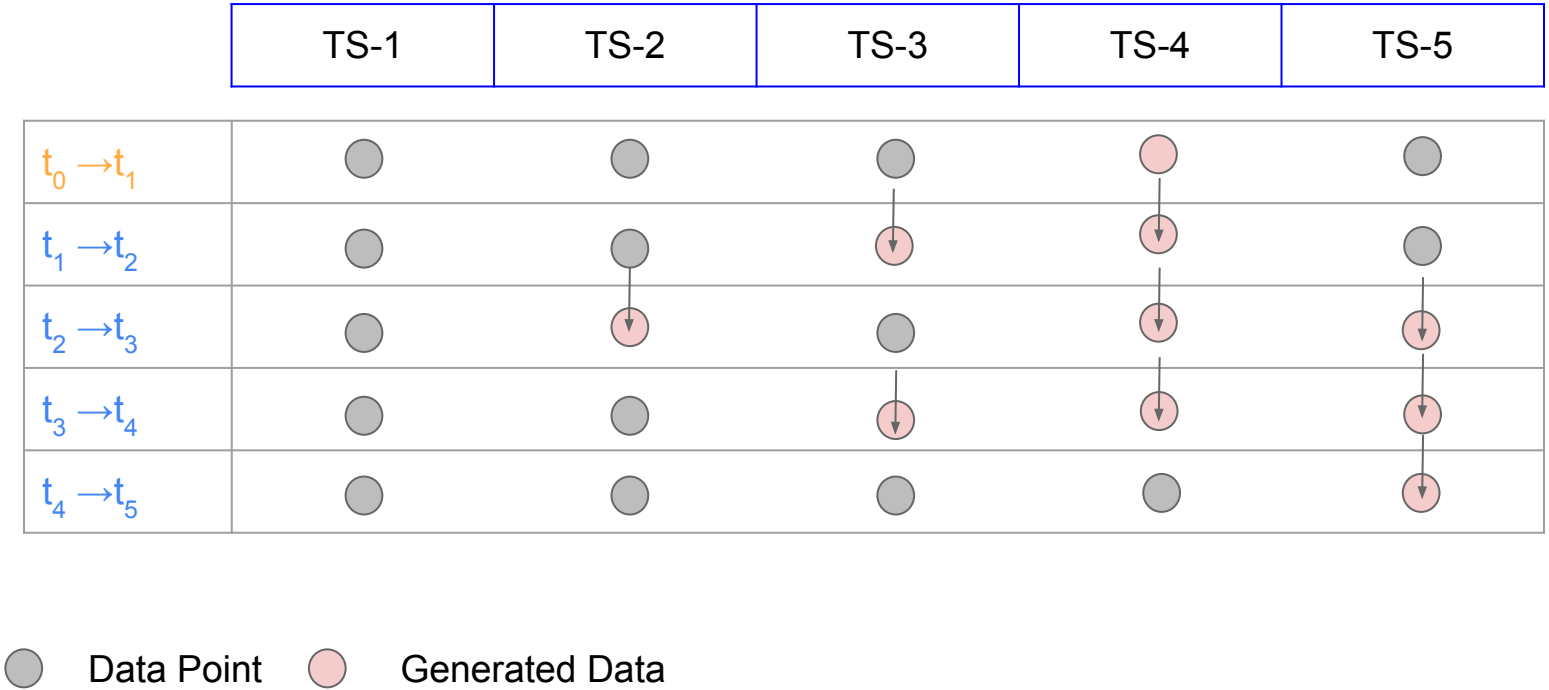
Looping timers - Advantages

- No need for a fanout operation.
- Timer will auto initiate when a key is seen for the first time.
- You can set a per key time to live.
  - Set a property with the key on the TTL of the looping timer.
  - Store the last seen timestamp value in state
  - Before rest of timers check if lastSeenTimestamp+TTL < OnTimer.timestamp
- OnTimer code is also where we can propagate last value seen

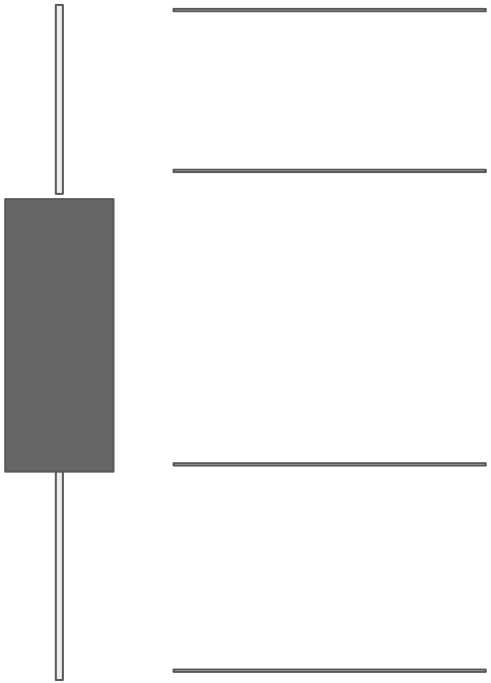
Gap filling


TS-1
TS-2
TS-3
TS-4
TS-5
$t_0 \rightarrow t_1$
$t_1 \rightarrow t_2$
$t_2 \rightarrow t_3$
$t_3 \rightarrow t_4$
$t_4 \rightarrow t_5$
Data Point
Generated Data

Gap filling - with hold and propagation



Concrete Example - Building candlesticks



*https://pixabay.com/photos/chart-trading-courses-forex-1905224/

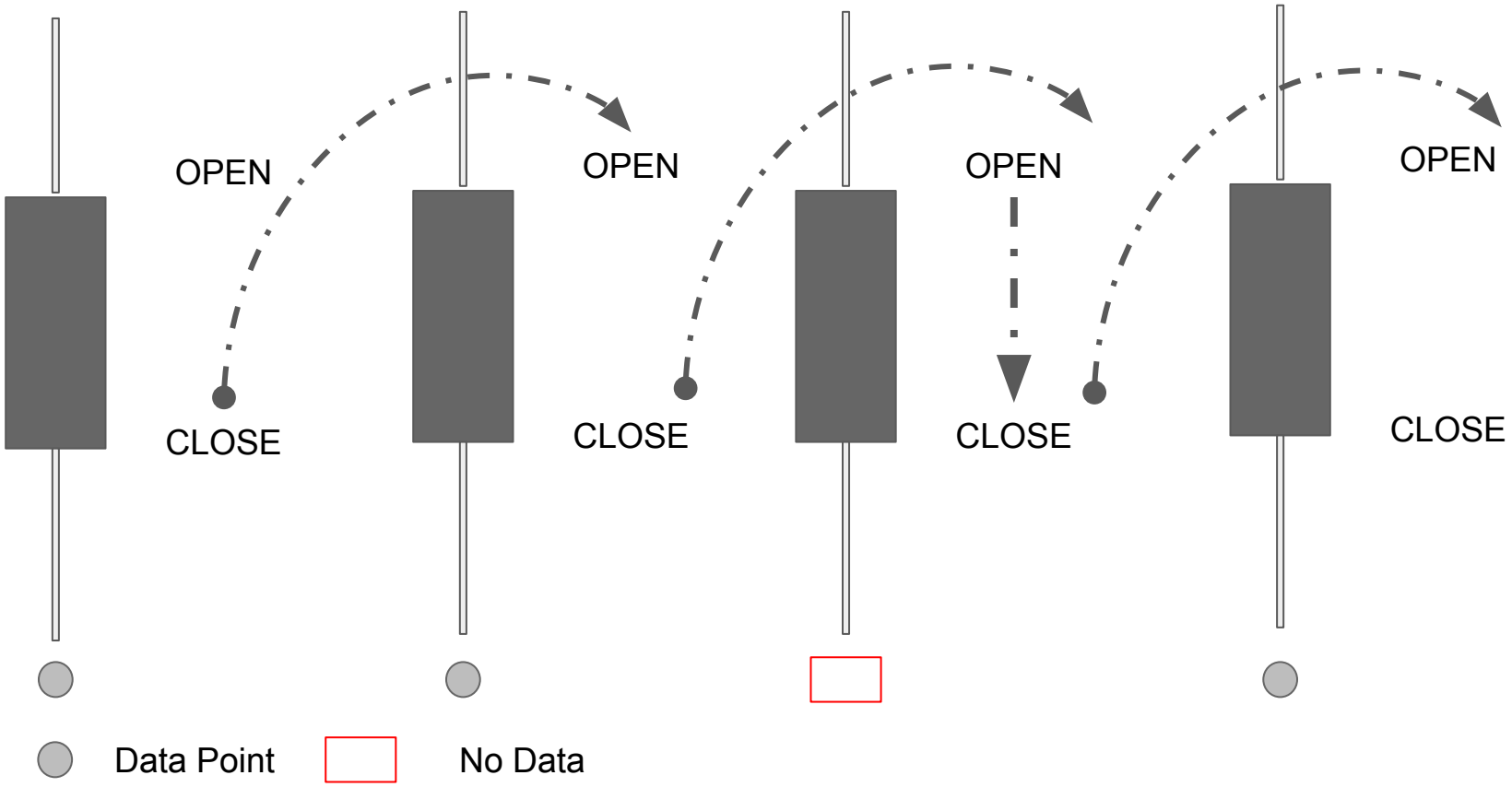
Example data

Time	Price Stock A	Price Stock B
14:00:00.000	1.54	0.98
14:00:00.500	1.53	-
14:00:01.200	-	0.99
14:00:02.200	1.53	-
14:00:03.100	1.52	1.00
14:00:03:300	1.52	-
14:00:03:400	-	0.98

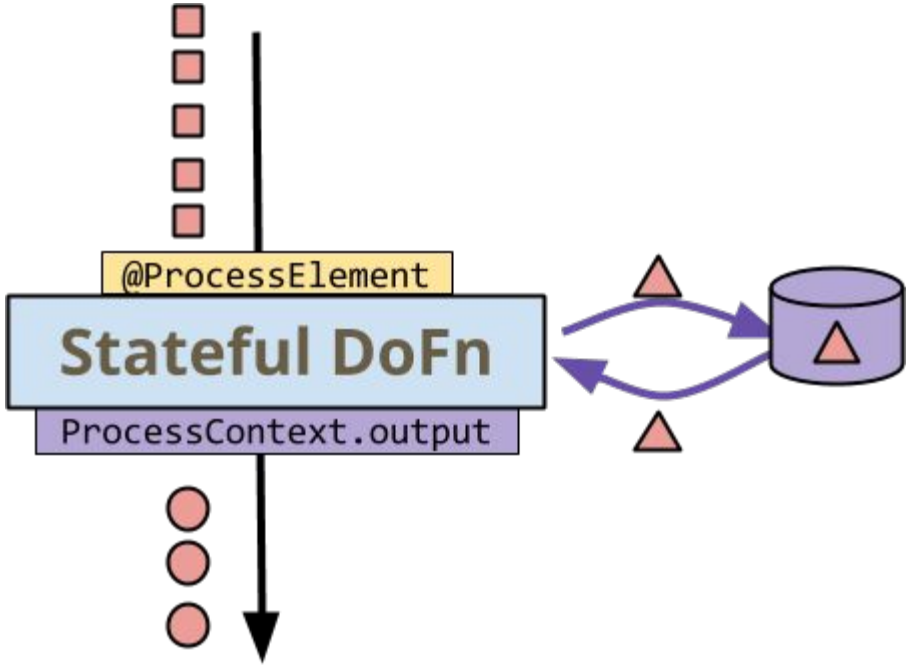
FixedWindow - GBK

Time	Price Stock A	Price Stock B
[14:00:00.000,14:00:00.999)	{1.54,1.53}	{0.98}
[14:00:01.000,14:00:01.999)	No Data	{0.99}
[14:00:02.000,14:00:02.999)	{1.53}	No Data
[14:00:03.000,14:00:03.999)	{1.52,1.52}	{1.00,0.98}

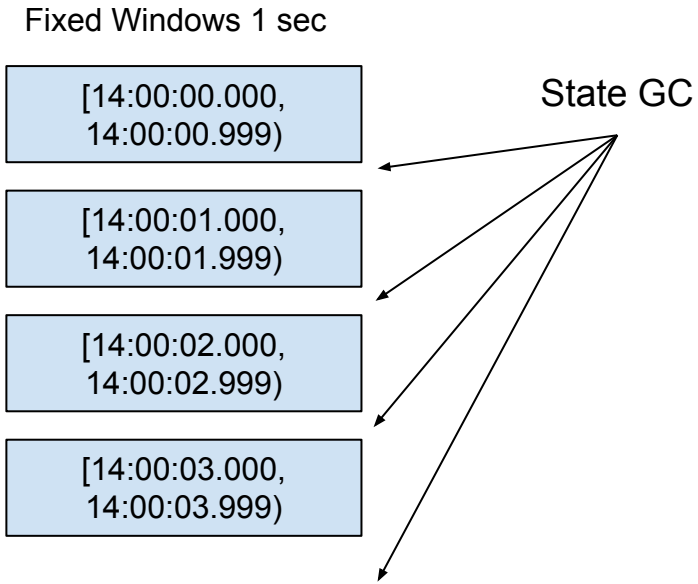
Concrete Example - Building candlesticks



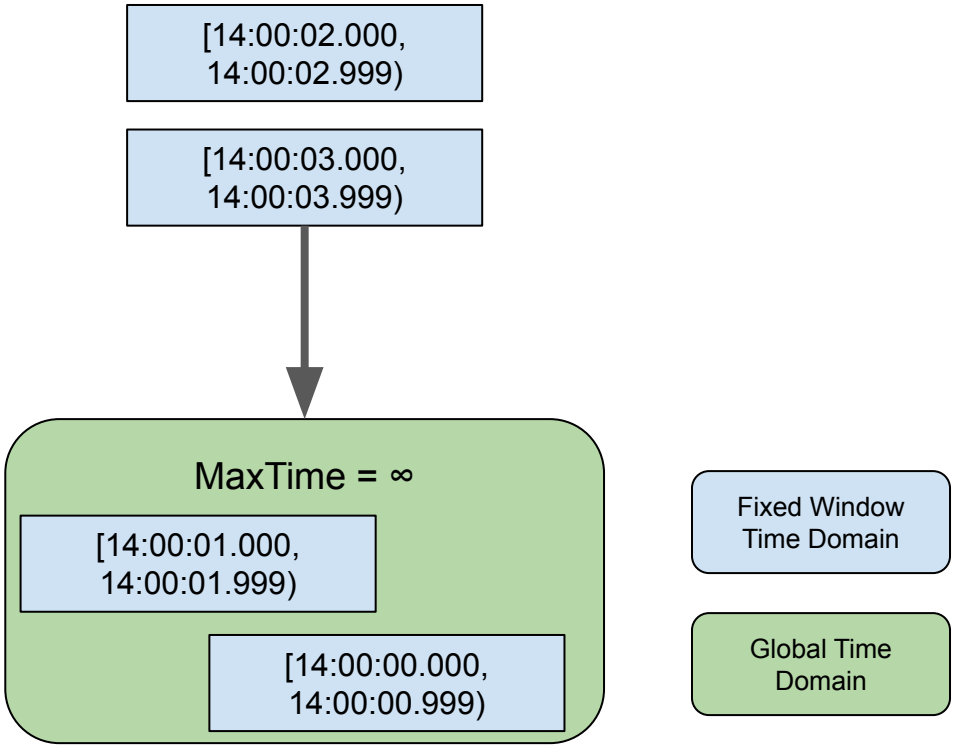
State API - Per Key-Window State



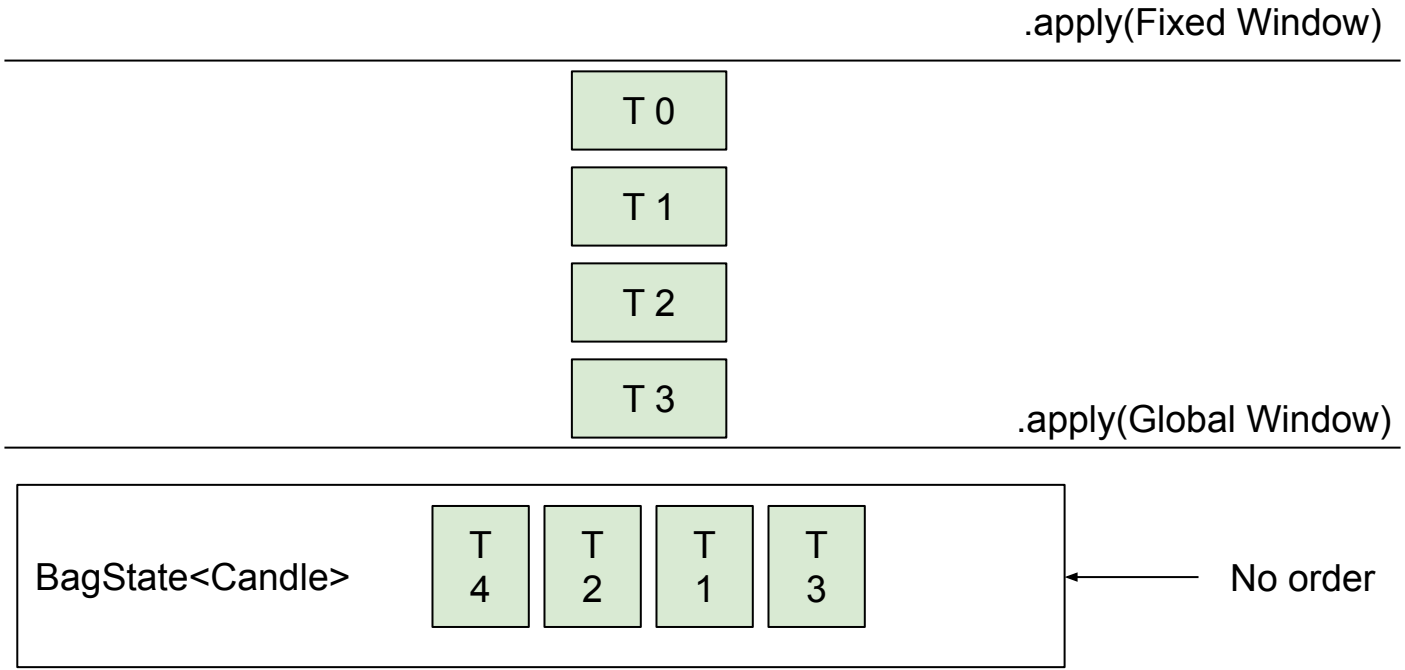
Transfer data across aggregations..



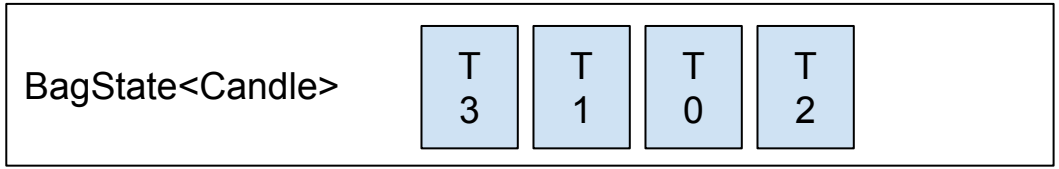
Global Windows ... no order!



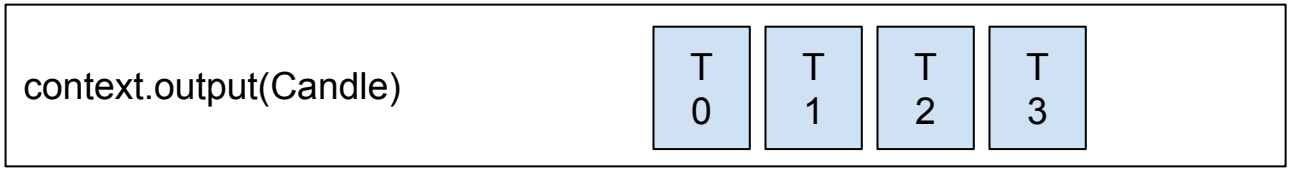
Dealing with lack of order



Dealing with lack of order



OnTimer (T0), OnTimer(T1),OnTimer(...) - Order and process



Interval list - Looping Timer + Global State

Time	Price Stock A	Price Stock B
[14:00:00.000,14:00:00.999)	{1.54,1.53}	{0.98}
[14:00:01.000,14:00:01.999)	Previous Close {1.53} No Data -	Previous Close {0.98} {0.99}
[14:00:02.000,14:00:02.999)	Previous Close {1.53} {1.53} -	Previous Close {0.99} No Data - 0.99
[14:00:03.000,14:00:03.999)	Previous Close {1.53} {1.52,1.52}	Previous Close {0.99} {1.00,0.98}

Final result and limitations

- Limitation 1
  - In memory sort of the key up to the OnTimer timestamp.
  - The accumulation step before is therefore important for data heavy loads. This will act as compression of the data before the global window step.
- Limitation 2
  - In order to GC the lists we need to do a full read-modify-write of whole list.
- Limitation 3
  - Late Data