CS 591 K1: Data Stream Processing and Analytics

Spring 2020

2/11: Windows and Triggers

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Window operators

- Practical way to perform operations on unbounded input
  - e.g. joins, holistic aggregates
- Compute on most *recent* events only
  - when providing real-time traffic information, you probably don't care about an accident that happened 2 hours ago
- *Recent* might mean different things
  - last 5 sec
  - last 10 events
  - last 1h every 10 min
  - last user session

Example: Window sensor readings

```
object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)

    val maxTemp = sensorData
      .map(r => Reading(r.id,r.time,(r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
     .timeWindow(Time.minutes(1))
      .max("temp")
  }
}
```


Vasiliki Kalavri | Boston University 2020

Configuring a time characteristic

In the DataStream API, you can use the time characteristic to tell Flink how to define time when you are creating windows. The time characteristic is a property of the `StreamExecutionEnvironment`:

```
object AverageSensorReadings {

    def main(args: Array[String]) {
    // set up the streaming execution environment
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // use event time for the application
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    // ingest sensor stream
    val sensorData: DataStream[SensorReading] = env.addSource(...)
    }
}
```

Or `ProcessingTIme` or `IngestionTime`

Keyed vs. non-keyed windows

Window operators can be applied on a keyed or a non-keyed stream:

- Window operators on keyed windows are evaluated *in parallel*
- Non-keyed windows are processed *in a single thread*

To create a window operator, you need to specify two window components:

- A **window assigner** determines how the elements of the input stream are grouped into windows. A window assigner produces a `WindowedStream` (or `All WindowedStream` if applied on a non-keyed `DataStream`).

- A **window function** is applied on a `WindowedStream` (or `AllWindowedStream`) and processes the elements assigned to a window.

Keyed vs. non-keyed windows

```
// define a keyed window operator
stream
.keyBy(...)
.window(...) // specify the window assigner
.reduce/aggregate/process(...) // specify the window function
```

```
// define a non-keyed window-all operator
stream
.windowAll(...) // specify the window assigner
.reduce/aggregate/process(...) // specify the window function
```

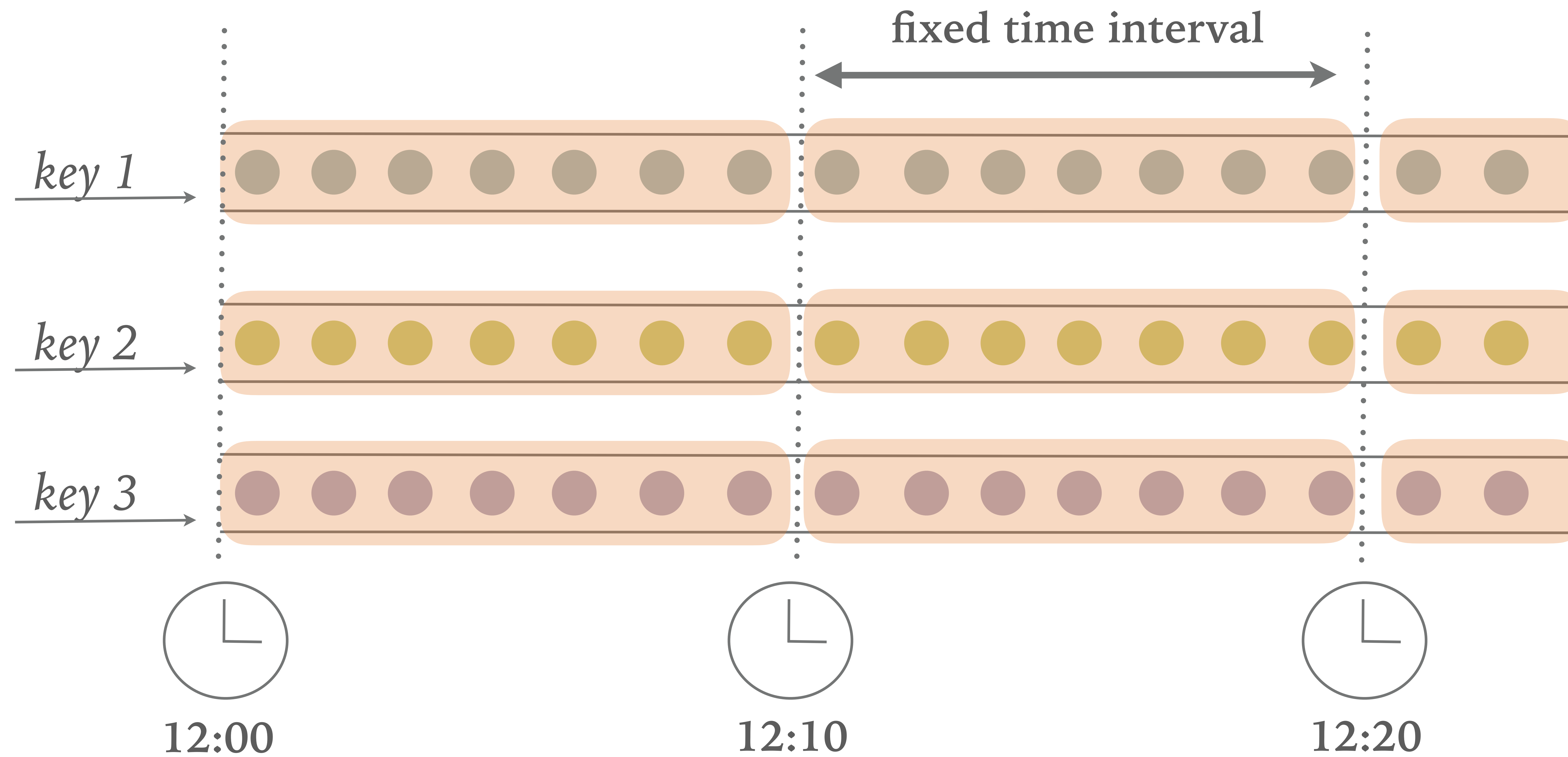
Built-in Window assigners in Flink

Time-based window assigners for the most common windowing use cases:

- They assign an element based on its event-time timestamp or the current processing time to windows.
- Time windows have a **start** and an **end** timestamp.
- All built-in window assigners provide **a default trigger** that triggers the evaluation of a window once the (processing or event) time passes the end of the window.
- A window is created when the first element is assigned to it. Flink will never evaluate empty windows!

Flink's built-in window assigners create windows of type `TimeWindow`. : a time interval between the two timestamps, where **start is inclusive** and **end is exclusive**.

Tumbling windows



non-overlapping buckets of fixed size

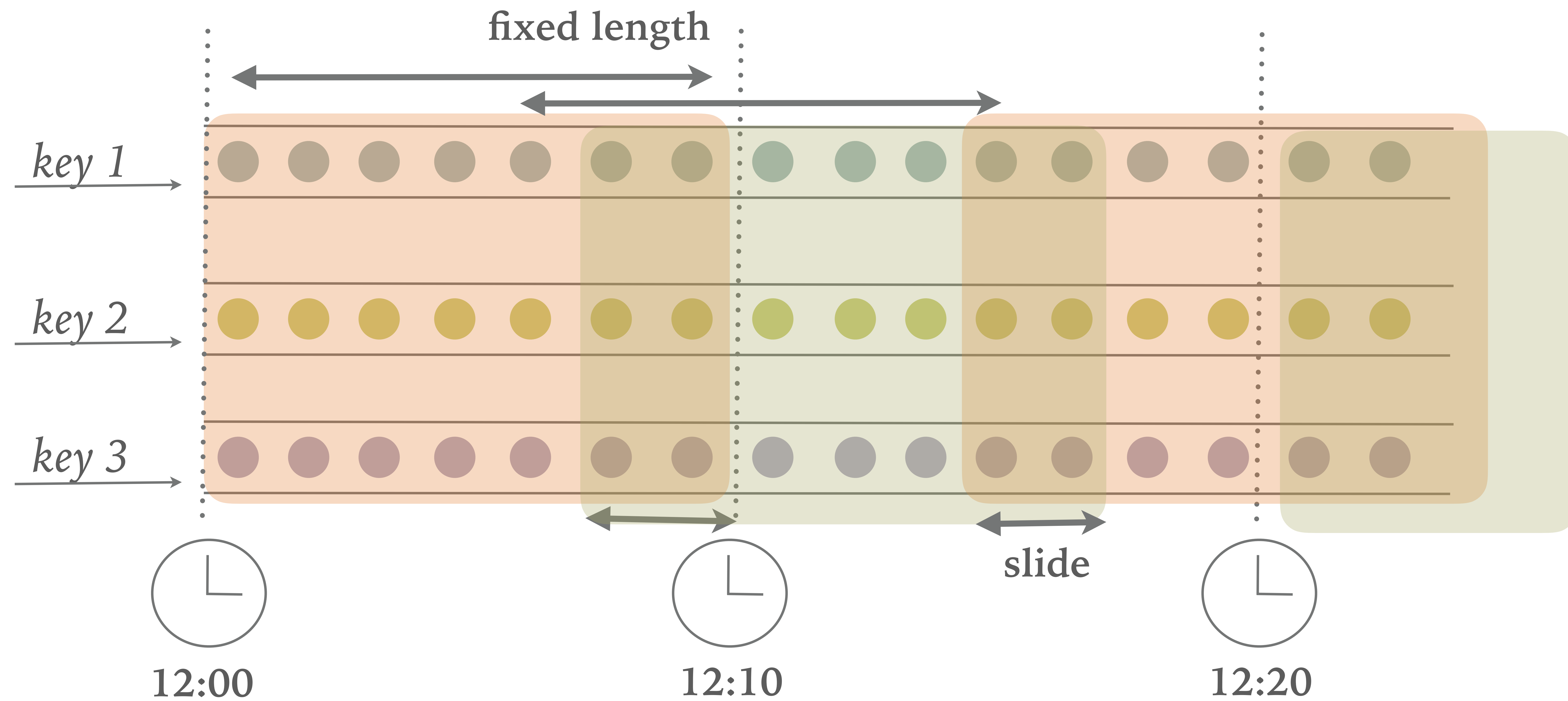
Tumbling window example

```
val sensorData: DataStream[SensorReading] = ...

val avgTemp = sensorData
     .keyBy(_.id)
     // group readings in 1s event-time windows
     .window(TumblingEventTimeWindows.of(Time.seconds(1)))
     .process(new TemperatureAverager)


val avgTemp = sensorData
     .keyBy(_.id)
     // shortcut for window.(TumblingEventTimeWindows.of(size))
     .timeWindow(Time.seconds(1))
     .process(new TemperatureAverager)
```

Window assigner

Window function

Sliding windows



overlapping buckets of fixed size

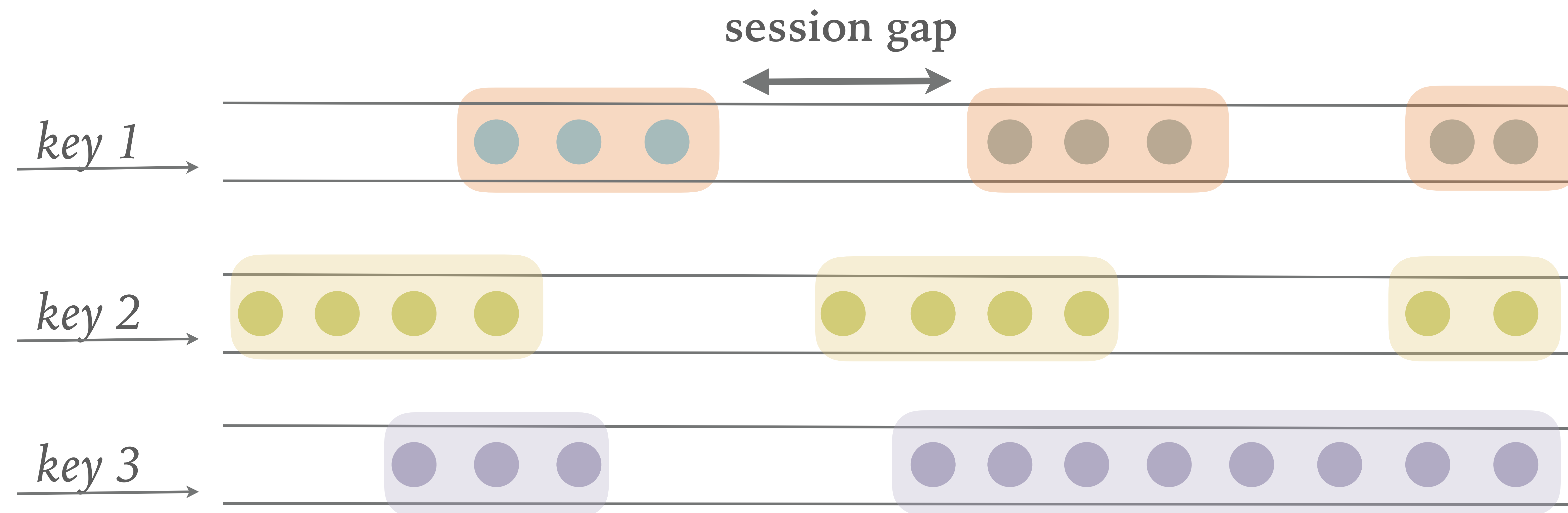
Sliding window example

```
val sensorData: DataStream[SensorReading] = ...

// event-time sliding windows assigner
val slidingAvgTemp = sensorData
    .keyBy(_.id)
    // create 1h event-time windows every 15 minutes
    .window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(15)))
    .process(new TemperatureAverager)


val slidingAvgTemp = sensorData
    .keyBy(_.id)
    // shortcut for window.(SlidingEventTimeWindow.of(size, slide))
    .timeWindow(Time.hours(1), Time(minutes(15)))
    .process(new TemperatureAverager)
```

Window assigner

Window function

Session windows



a period of *activity* followed by a period of *inactivity*

Session window example

```
// event-time session windows assigner
val sessionWindows = sensorData
    .keyBy(_.id)
    // create event-time session windows with a 15 min gap
    .window(EventTimeSessionWindows.withGap(Time.minutes(15)))
    .process(…)
```

Window assigner

Window function

Window functions

Window functions define the computation that is performed on the elements of a window

- **Incremental aggregation functions** are applied when an element is added to a window:
  - They maintain a single value as window state and eventually emit the aggregated value as the result.
  - `ReduceFunction` and `AggregateFunction`
- **Full window functions** collect all elements of a window and iterate over the list of all collected elements when evaluated:
  - They require more space but support more complex logic.
  - `ProcessWindowFunction`

ReduceFunction example

```
val minTempPerWindow: DataStream[(String, Double)] = sensorData
    .map(r => (r.id, r.temperature))
    .keyBy(_._1)
    .timeWindow(Time.seconds(15))
    .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

The function is evaluated for every element entering the window

Input and output types must be the same

AggregateFunction interface

```
public interface AggregateFunction<IN, ACC, OUT> extends Function,
Serializable {

    // create a new accumulator to start a new aggregate.
    ACC createAccumulator();

    // add an input element to the accumulator and return the accumulator.
    ACC add(IN value, ACC accumulator);

    // compute the result from the accumulator and return it.
    OUT getResult(ACC accumulator);

    // merge two accumulators and return the result.
    ACC merge(ACC a, ACC b);

}
```

AggregateFunction example

```scala
val avgTempPerWindow: DataStream[(String, Double)] = sensorData
    .map(r => (r.id, r.temperature))
    .keyBy(_._1)
    .timeWindow(Time.seconds(15))
    .aggregate(new AvgTempFunction)

// An AggregateFunction to compute the average temperature per sensor.
// The accumulator holds the sum of temperatures and an event count.
class AvgTempFunction extends AggregateFunction[(String, Double), (String, Double, Int), (String, Double)]
{
    override def createAccumulator() = { ("", 0.0, 0)}

    override def add(in: (String, Double), acc: (String, Double, Int)) = {
        (in._1, in._2 + acc._2, 1 + acc._3)
    }

    override def getResult(acc: (String, Double, Int)) = { (acc._1, acc._2 / acc._3) }

    override def merge(acc1: (String, Double, Int), acc2: (String, Double, Int)) = {
        (acc1._1, acc1._2 + acc2._2, acc1._3 + acc2._3)
    }
}
```

Input type

Accumulator type

Output type

Initialization

Accumulate one element

Compute the result

Merge two partial accumulators

ProcessWindowFunction

Use the `ProcessWindowFunction` to perform arbitrary computations on the contents of a window:

- The `process()` method is called with the key of the window, an `Iterable` to access the elements of the window, and a `Collector` to emit results.
- A `Context` gives access to the metadata of the window (start and end timestamps in the case of a time window), the current processing time and the watermark.

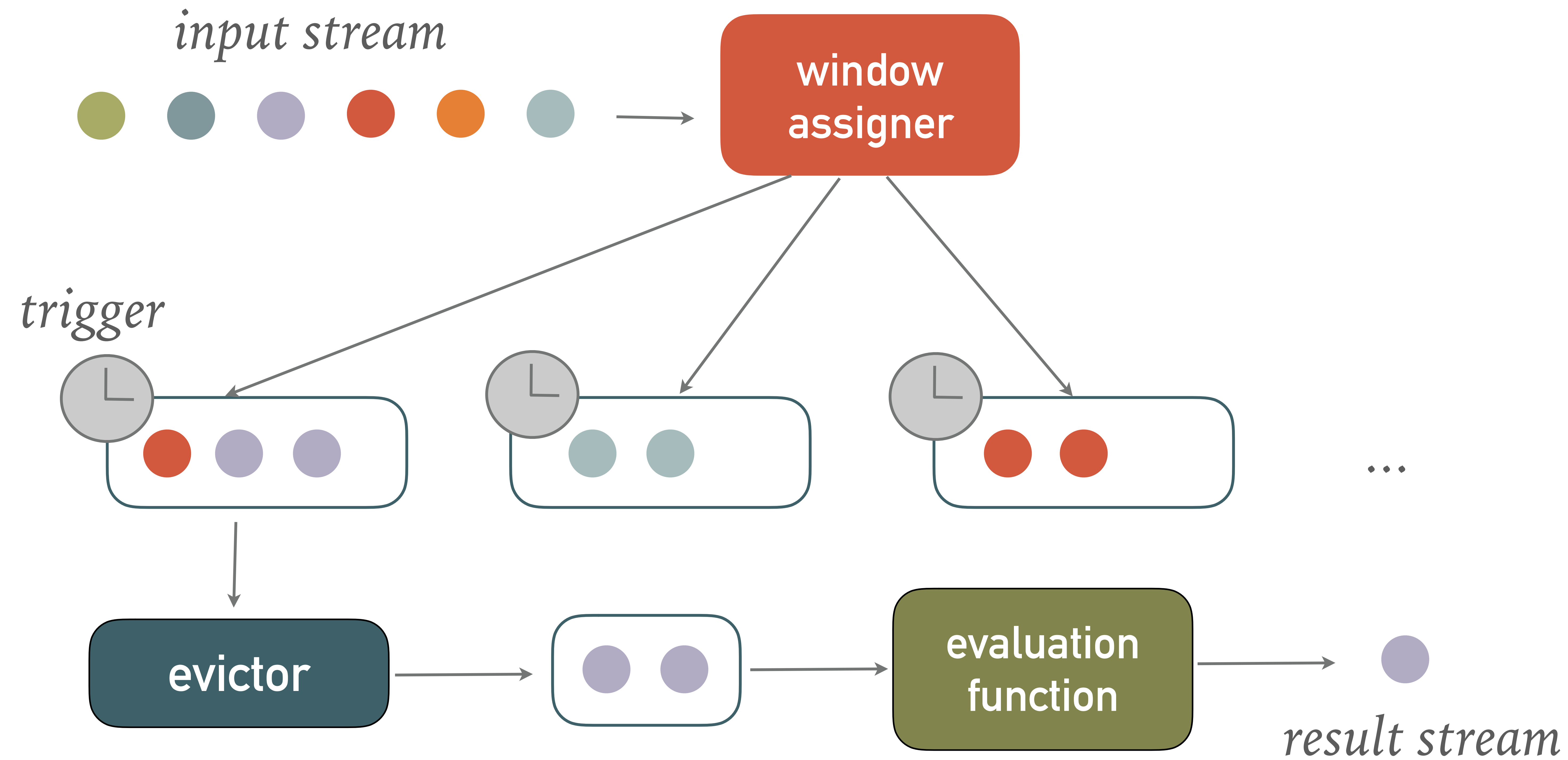
ProcessWindowFunction interface

```
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
    extends AbstractRichFunction {

    // Evaluates the window
    void process(
        KEY key, Context ctx, Iterable<IN> vals, Collector<OUT> out) throws Exception;


    public abstract class Context implements Serializable {

        // Returns the metadata of the window
        public abstract W window();

        // Returns the current processing time
        public abstract long currentProcessingTime();

        // Returns the current event-time watermark
        public abstract long currentWatermark();
    }
}
```

Iterate over the window contents

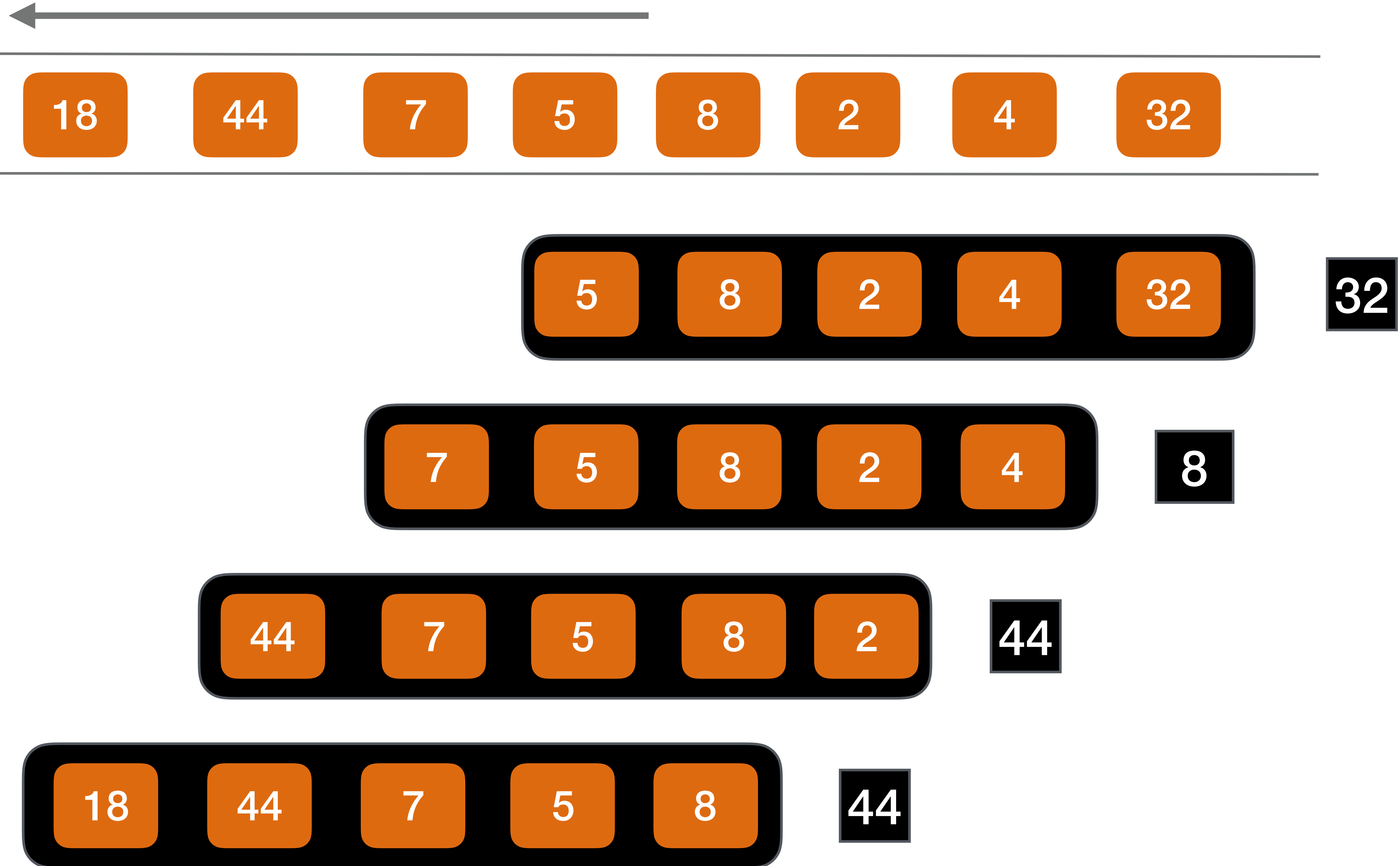
Get start and end timestamps

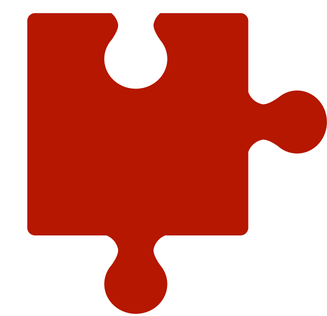
Custom windows



- Describe each component

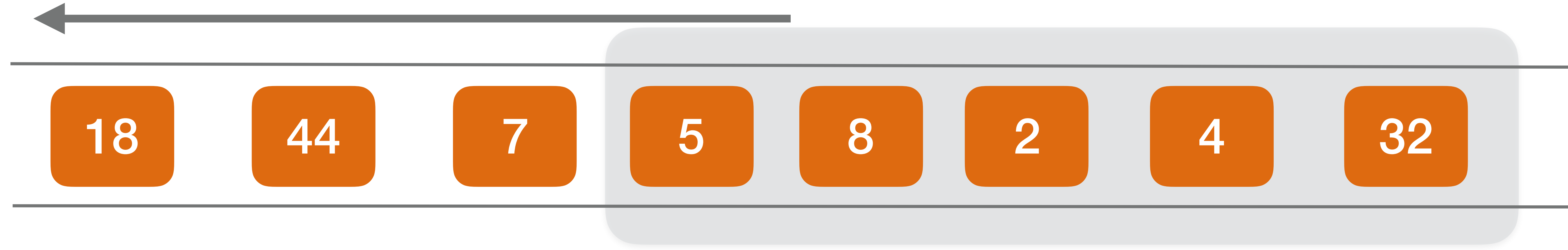
Window max over 5 last elements?





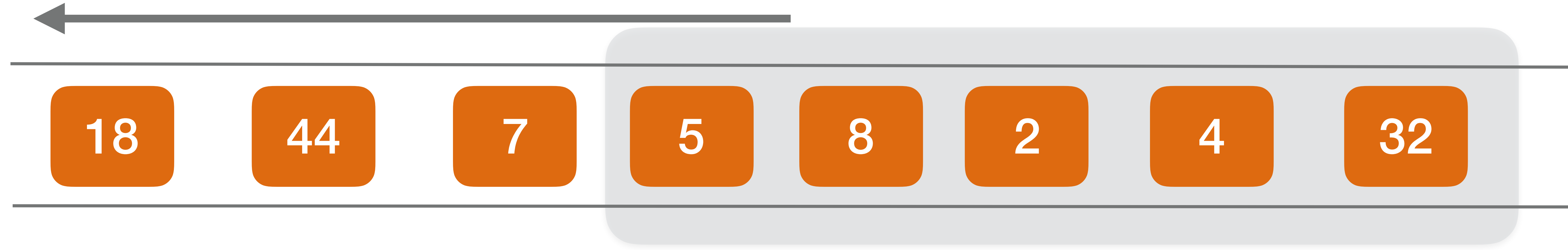
Can we compute the max more efficiently?

Window max over 5 last elements?

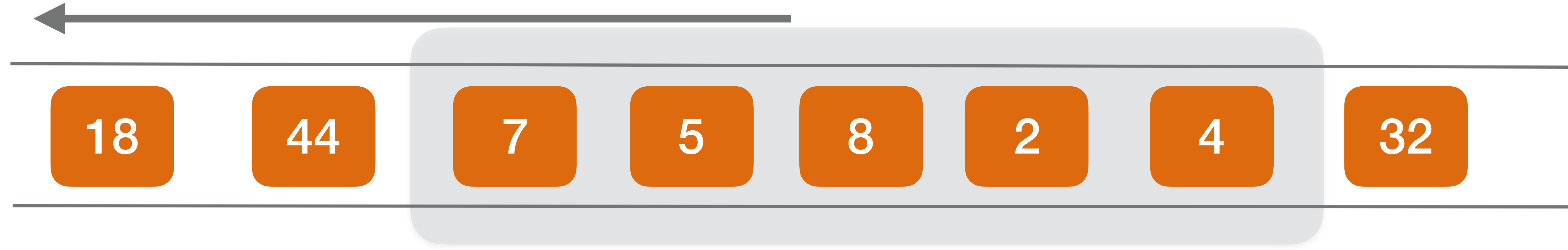


inwindow

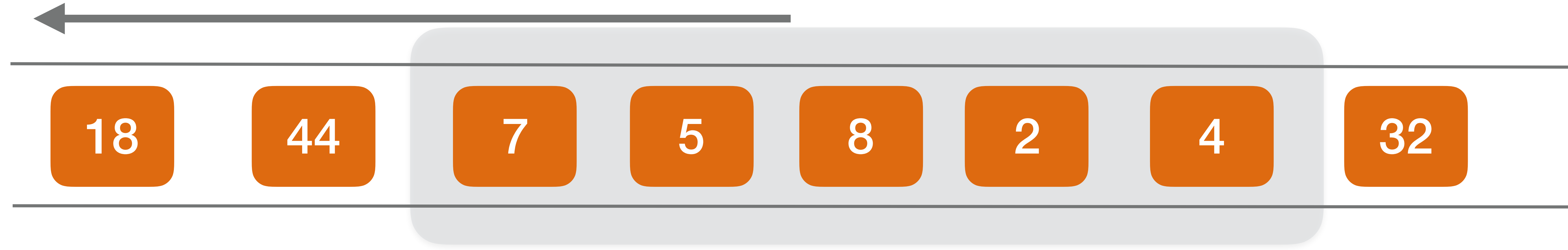
Window max over 5 last elements?



Window max over 5 last elements?



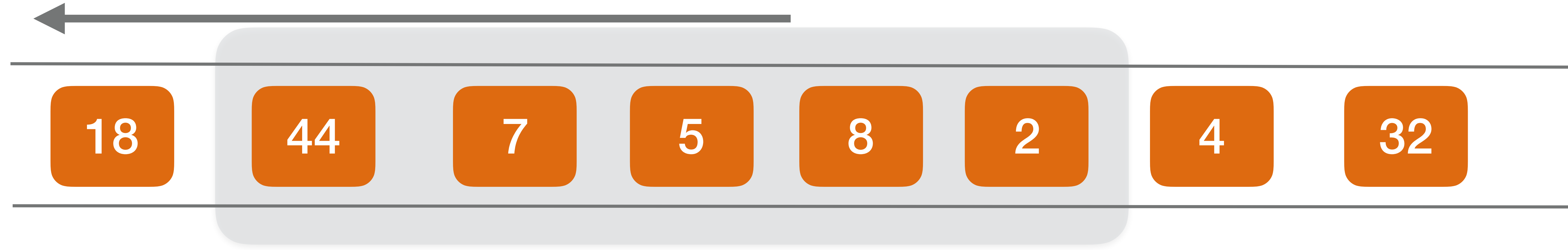
Window max over 5 last elements?



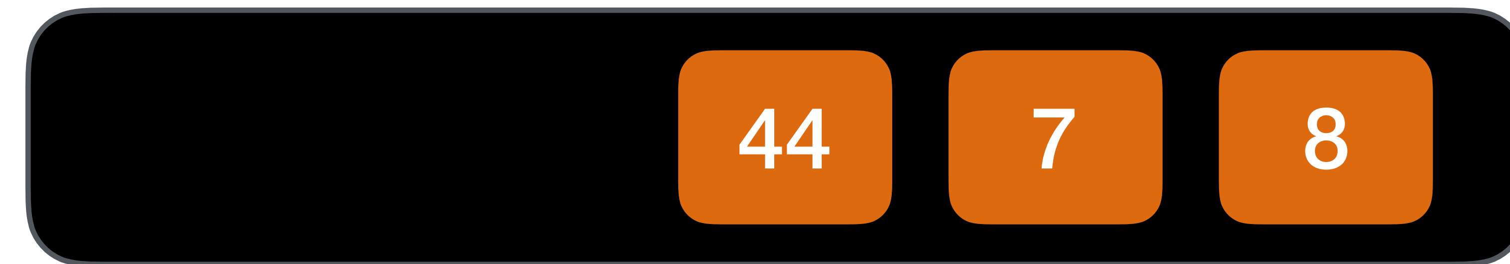
inwindow

8

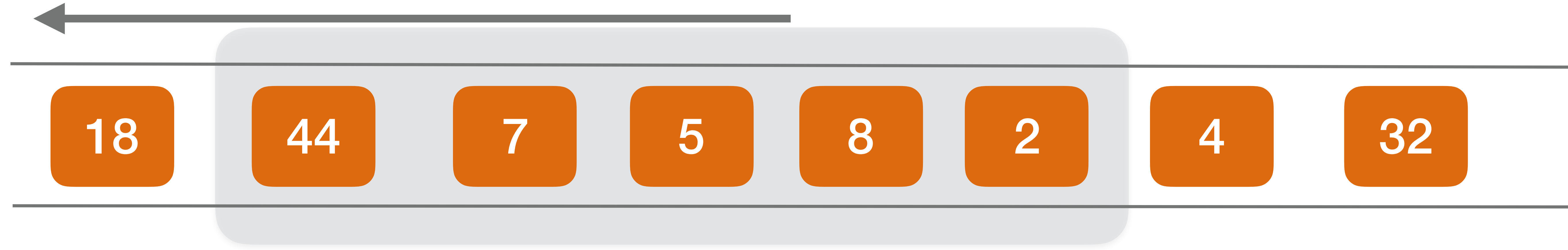
Window max over 5 last elements?



inwindow



Window max over 5 last elements?



and so on…

Process Functions

Advanced transformation functions used to implement custom logic for which predefined windows and transformations might not be suitable:

- they provide access to record timestamps and watermarks
- they can register timers that trigger at a specific time in the future

`ProcessFunction`, `KeyedProcessFunction`, `CoProcessFunction`, `ProcessJoinFunction`, `BroadcastProcessFunction`, `KeyedBroadcastProcessFunction`, `ProcessWindowFunction`, and `ProcessAllWindowFunction`.

KeyedProcessFunction

The `KeyedProcessFunction` is applied to a `KeyedStream`:

- **`processElement(v: IN, ctx: Context, out: Collector[OUT])`** is called for each record of the stream. Result records are emitted by passing them to the `Collector`. The `Context` object gives access to the timestamp and the key of the current record and to a `TimerService`.

- **`onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT])`** is invoked when a previously registered timer triggers. The timestamp argument gives the timestamp of the firing timer and the `Collector` allows emitting records. The `OnTimerContext` provides the same services as the `Context` object of the `processElement()` method and also returns the time domain (processing time or event time) of the firing timer.

Vasiliki Kalavri | Boston University 2020

ProcessFunction example

```
val warnings = readings
    .keyBy(_.id) // key by sensor id
    .process(new TempIncreaseAlertFunction) // apply ProcessFunction to monitor temperatures

/** Emits a warning if the temperature of a sensor monotonically increases for 1 second (in processing time) **/
class TempIncreaseAlertFunction extends KeyedProcessFunction[String, SensorReading, String] {

    // stores temperature of last sensor reading
    val lastTemp: Double
    // stores timestamp of currently active timer
    val currentTimer: Long

    override def processElement(r: SensorReading, ctx:Context, out: Collector[String]): Unit = {
        // get previous temperature
        val prevTemp = lastTemp
        // update last temperature
        lastTemp = r.temperature

        if (prevTemp == 0.0 || r.temperature < prevTemp) {
            // temperature decreased; delete current timer
            ctx.timerService().deleteProcessingTimeTimer(curTimer)
        } else if (r.temperature > prevTemp && curTimerTimestamp == 0) {
            // temperature increased and we have not set a timer yet: set processing time timer for now + 1 second
            val timerTs = ctx.timerService().currentProcessingTime() + 1000
            ctx.timerService().registerProcessingTimeTimer(timerTs)
        // remember current timer
        currentTimer = timerTs
        }
    }
}
```

onTimer() example

```
override def onTimer(
    ts: Long,
    ctx: OnTimerContext,
    out: Collector[String]): Unit = {

        out.collect("Temperature of sensor '" + ctx.getCurrentKey +
            "' monotonically increased for 1 second.")

        currentTimer.clear()
    }
}
```

CoProcess Function

For low-level operations on two inputs:

- One transformation method for each input `processElement1()` and `processElement2()`
- Both methods are called with a `Context` object that gives access to the element or timer timestamp and a `TimerService`
- You can use it to register timers and it provides an `onTimer()` callback method

CoProcessFunction example

```
val forwardedReadings = readings
    // connect readings and switches
    .connect(filterSwitches)
    // key by sensor ids
    .keyBy(_.id, _._1)
    // apply filtering CoProcessFunction
    .process(new ReadingFilter)
```

Key for the readings stream

Key for the filterSwitches stream

CoProcessFunction example

```
class ReadingFilter
extends CoProcessFunction[SensorReading, (String, Long), SensorReading] {

    // process readings
    override def processElement1(reading: SensorReading,ctx: Context, out:
        Collector[SensorReading]): Unit = {…}

    // process switches
    override def processElement2(switch: (String, Long),ctx: Context, out:
        Collector[SensorReading]): Unit = {…}

    // process timers
    override def onTimer(ts: Long, ctx: OnTimerContext, out:
        Collector[SensorReading]): Unit = {…}
}
```

Further reading

- “Introducing Stream Windows in Apache Flink”: https://flink.apache.org/news/2015/12/04/Introducing-windows.html
- “How Apache Flink™ Enables New Streaming Applications, Part 3”: https://www.ververica.com/blog/session-windowing-in-flink