CS 591 K1:
Data Stream Processing and Analytics

Spring 2021

Window aggregation

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Window operators

Window operators

- Practical way to perform operations on unbounded input
  - e.g. joins, holistic aggregates

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

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

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.

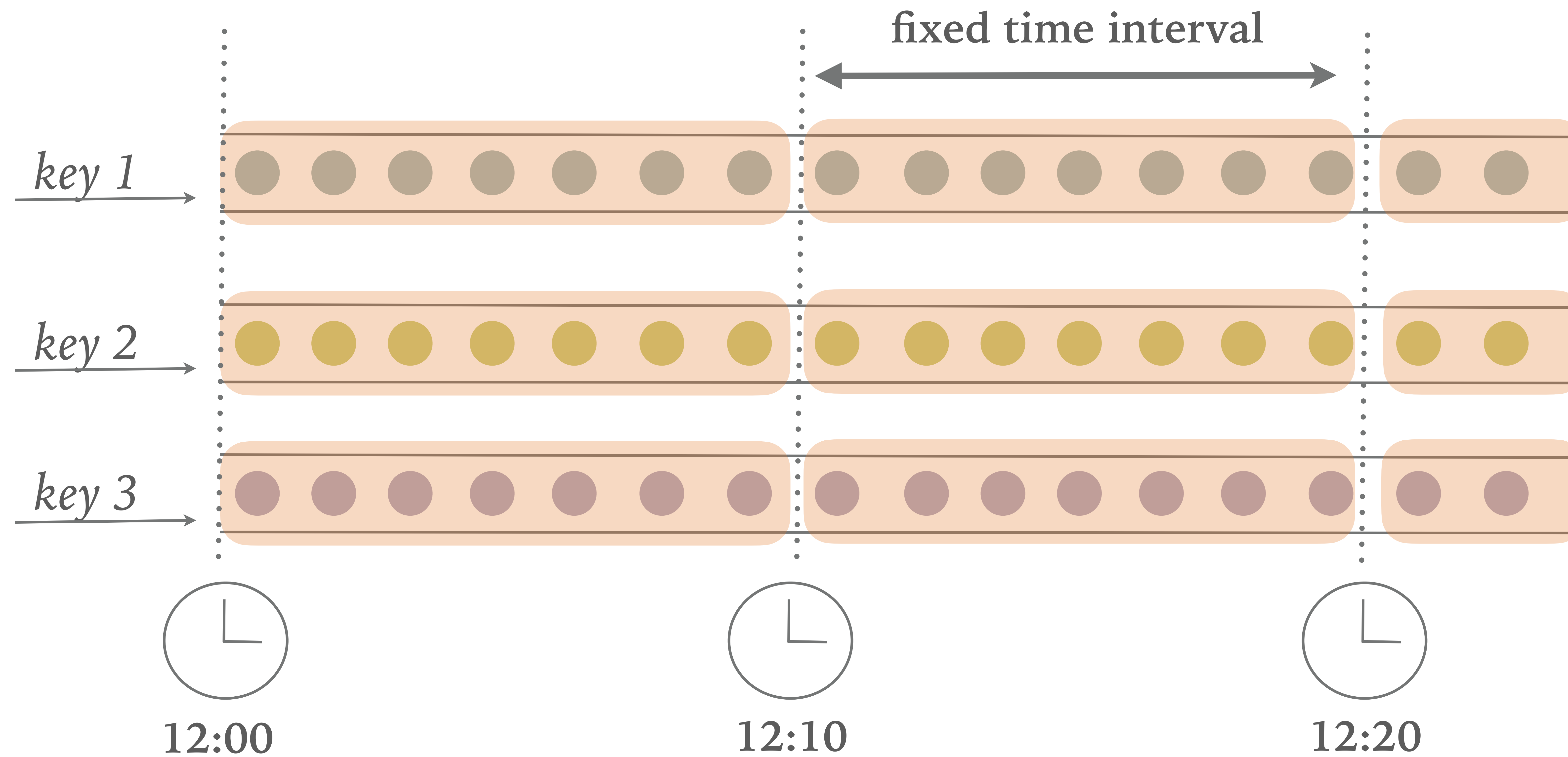
- A **window function** is applied on the window contents and processes the elements assigned to each window.

Window types

The window type defines the logic based on which we derive finite windows from a continuous stream of events:

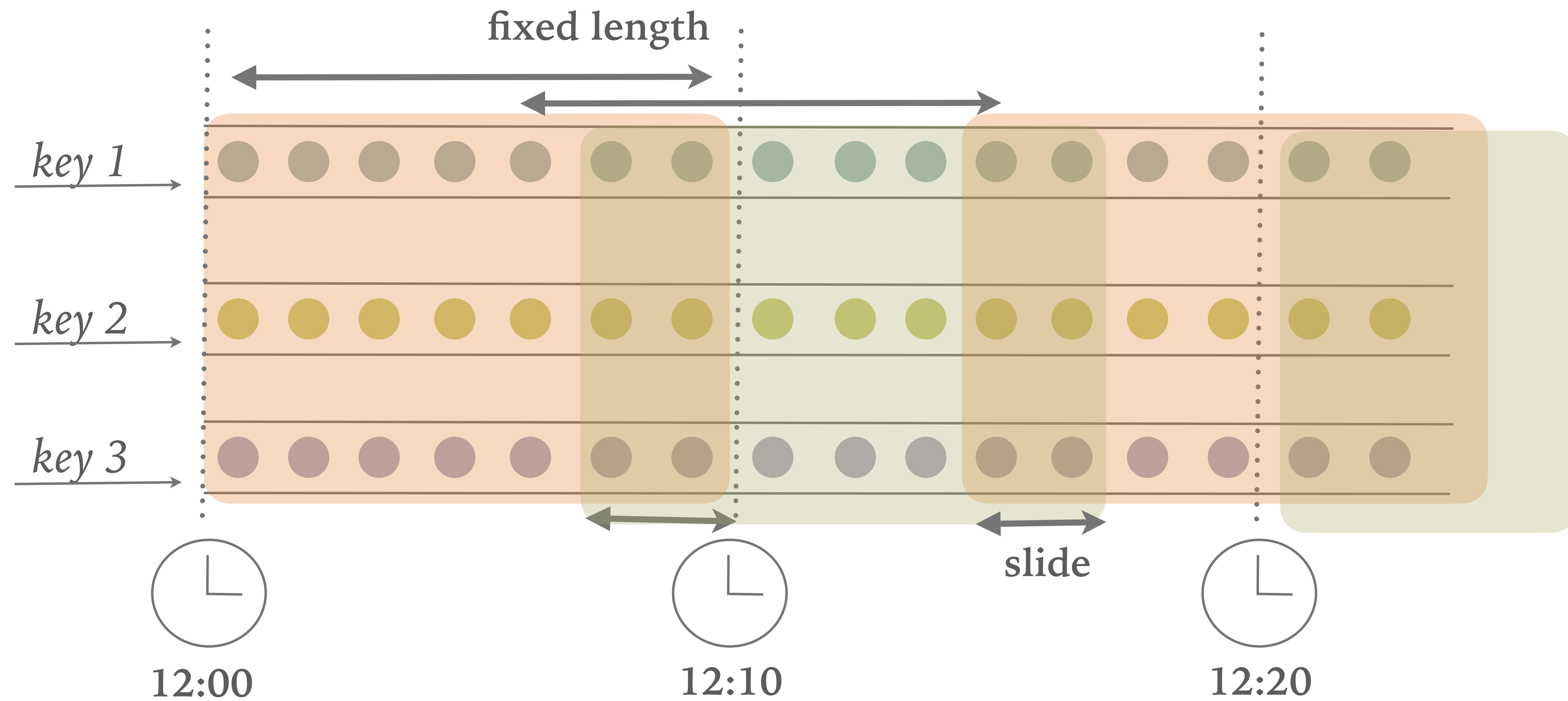
- **Tumbling** (fixed) windows split time into segments of equal length l. The end of a window marks the start of the next one.
  - Each event belongs to one window only.
- **Sliding** (hopping) windows further define a slide parameter ls which determines how often a new window starts. Consecutive windows overlap when $l_s < l$.
  - Events may belong to multiple windows.
- **Session** windows define a period of *activity* followed by a period of *inactivity*. A session window ends if no event arrives for some time gap l_g.

Tumbling windows



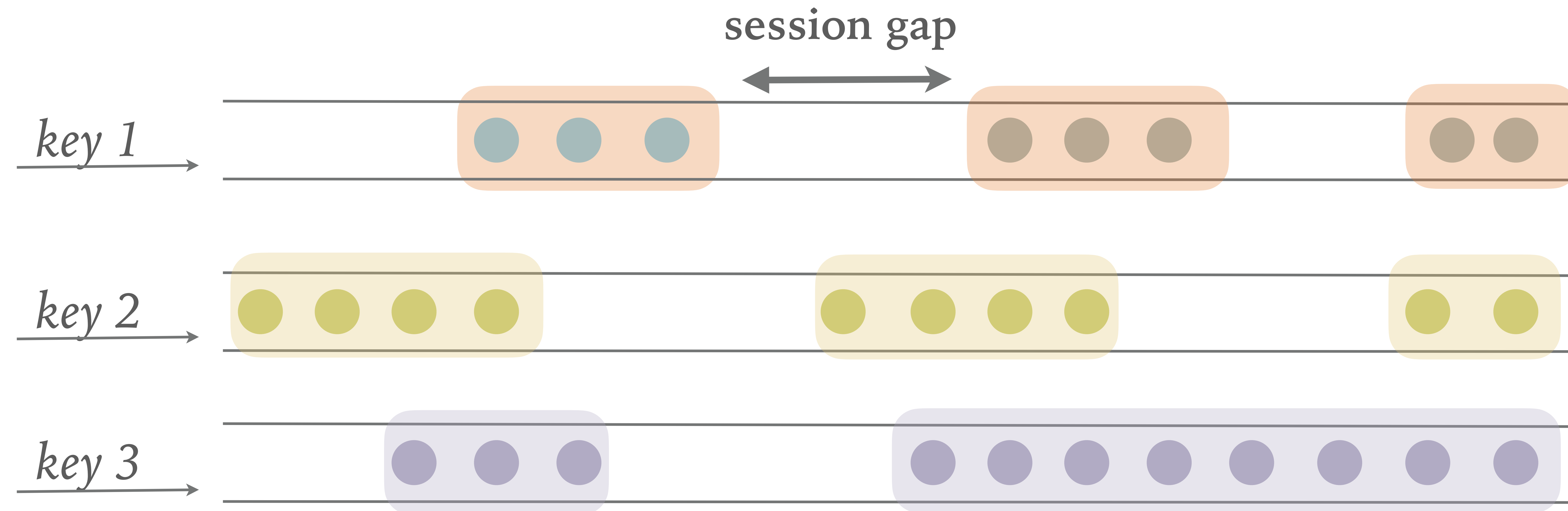
non-overlapping buckets of fixed size

Sliding windows



overlapping buckets of fixed size

Session windows



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

Windowing measures

We can define windows based on multiple (monotonically increasing) measures:

- **Time**, such as event or processing time
- **Count**, i.e. number of events
- **Data-dependent** advancing measure, such as a punctuation or other signal in the stream
  - e.g. when the amount of bids placed for an action exceeds a threshold

We can mix windowing measures to define multi-measure windows

- e.g. output the last 10 tuples (count) every 5 second (time).

Window aggregation functions

Window aggregation functions define the computation we perform on the elements of a window.

- **Distributive** functions: final values can be computed as the aggregation of partial aggregates with constant size.
  - min, max, sum
- **Algebraic** functions: final values can be computed by applying a function on partial aggregates of fixed size.
  - average, N largest values
- **Holistic** functions: partial aggregates have an unbounded size
  - median, most frequent, rank

Window context

We divide windows into 3 classes with regard to the context we need in order to know where windows start and end:

- **Context Free (CF)** windows are those for which we can compute their boundaries without processing any tuples.
    - Tumbling and sliding windows are context-free as we can compute all start and end timestamps based on their length and slide parameters.
- **Forward Context Free (FCF)** windows are those which depend on punctuations. We can compute their boundaries once we have processed all events up to a timestamp t.
- **Forward Context Aware (FCA)** windows require us to process tuples after timestamp t in order to compute all window boundaries before t.
    - Multi-measure windows are FCA.

🤧😷🤒 Vasiliki Kalavri | Boston University 2021

Window evaluation

Window evaluation strategies

A window operator is responsible for grouping incoming records into windows and making the evaluation function results available in the output whenever a window **triggers**, i.e. when the system's notion of time arrives at its end timestamp.

Evaluation functions can be applied **eagerly**, upon receiving a new record that belongs to the window, or **lazily**, on trigger.

Regardless of the strategy used, the operator performs two types of processing:

(i) upon receiving a new event in its input, the operator needs decide how to assign the event to one or more windows and possibly apply some partial aggregation.

(ii) upon receiving a trigger, the operator needs to decide which windows are "complete", apply the final aggregation, and produce results.

Record buffer

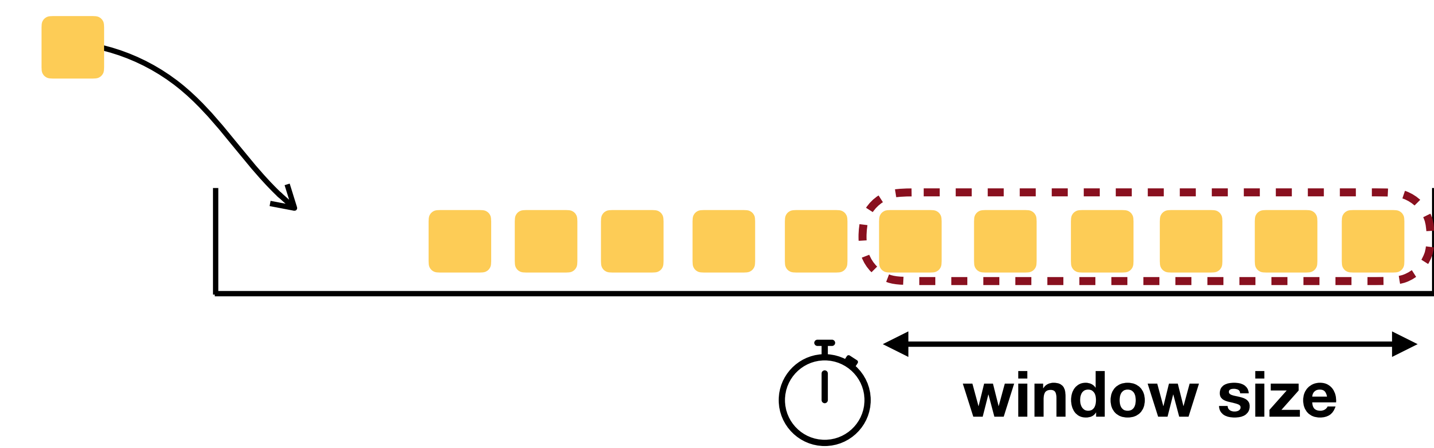
Use a buffer to store incoming events, ordered by timestamp.

On event: Append to the end of the buffer.

On trigger: Retrieve all records whose timestamp falls inside the window bounds.

The number of records in the buffer is proportional to the window size and input rate, thus, state requirements can grow significantly for high input rates and large windows.

The evaluation function is applied to the window contents lazily at trigger time.



Record buffer

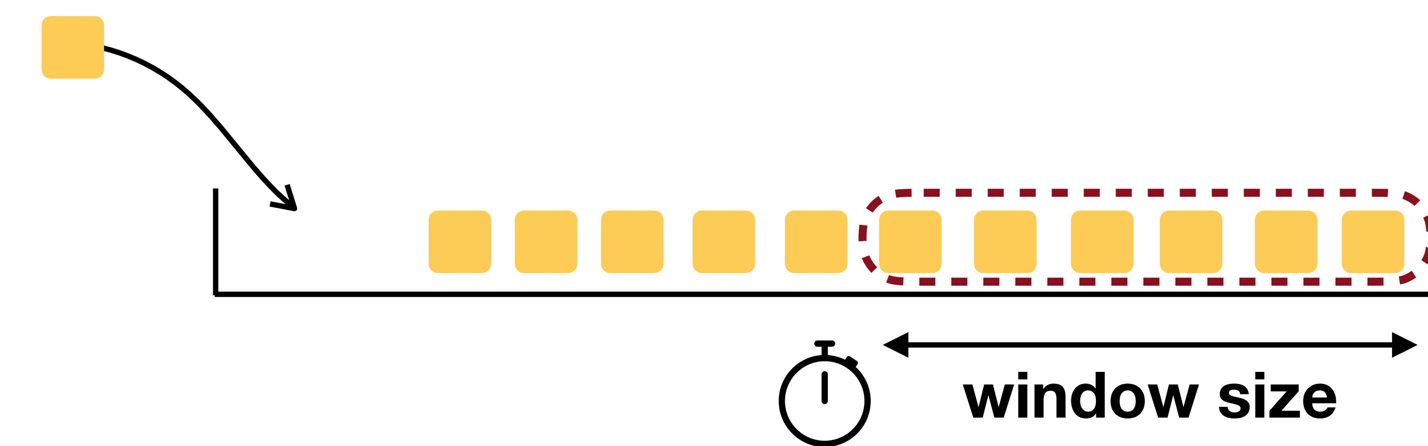
Use a buffer to store incoming events, ordered by timestamp.

On event: Append to the end of the buffer.

On trigger: Retrieve all records whose timestamp falls inside the window bounds.

The number of records in the buffer is proportional to the window size and input rate, thus, state requirements can grow significantly for high input rates and large windows.

The evaluation function is applied to the window contents lazily at trigger time.



- General strategy which can support all window types and aggregation functions.
- Evaluation can be inefficient for out-of-order streams (memory copies), high rate (increasing state), and overlapping windows (multiple aggregate computations).

Aggregate tree

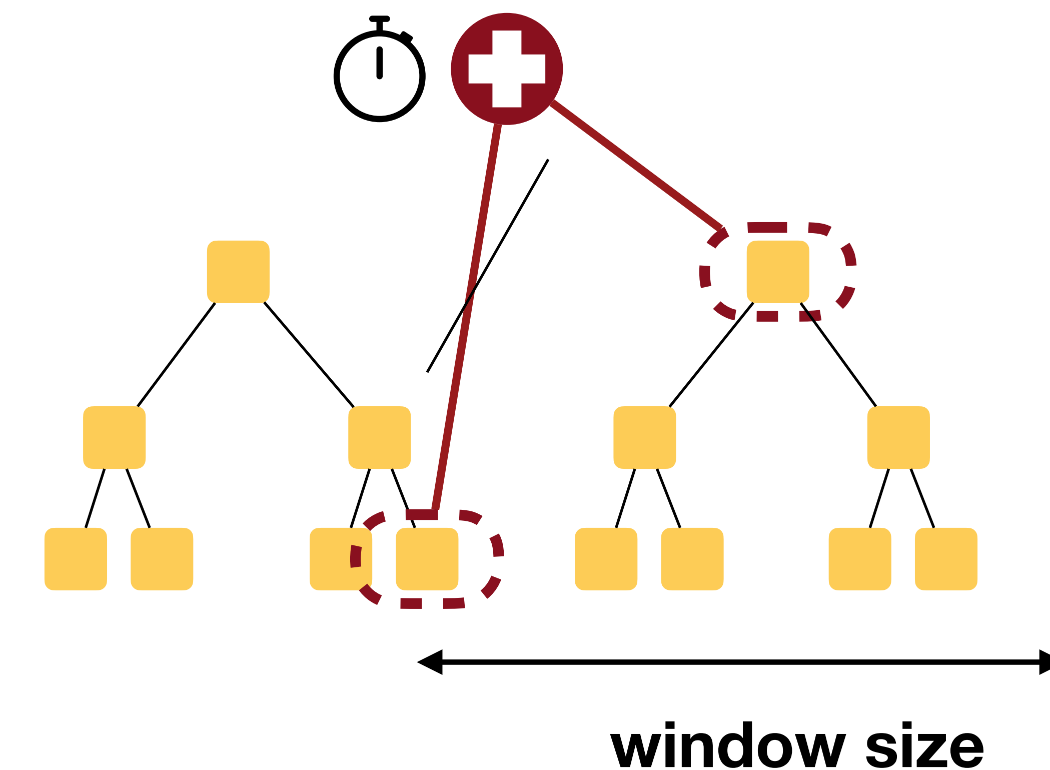
Use a binary tree to store partial aggregates on top of stream events.

On event: Insert to the binary tree and update all affected partial aggregates.

On trigger: Combine partial aggregated to compute and emit the final value.

The state requirements are high as both events and partial aggregates are maintained.

The evaluation function is applied on event and on trigger.



Aggregate tree

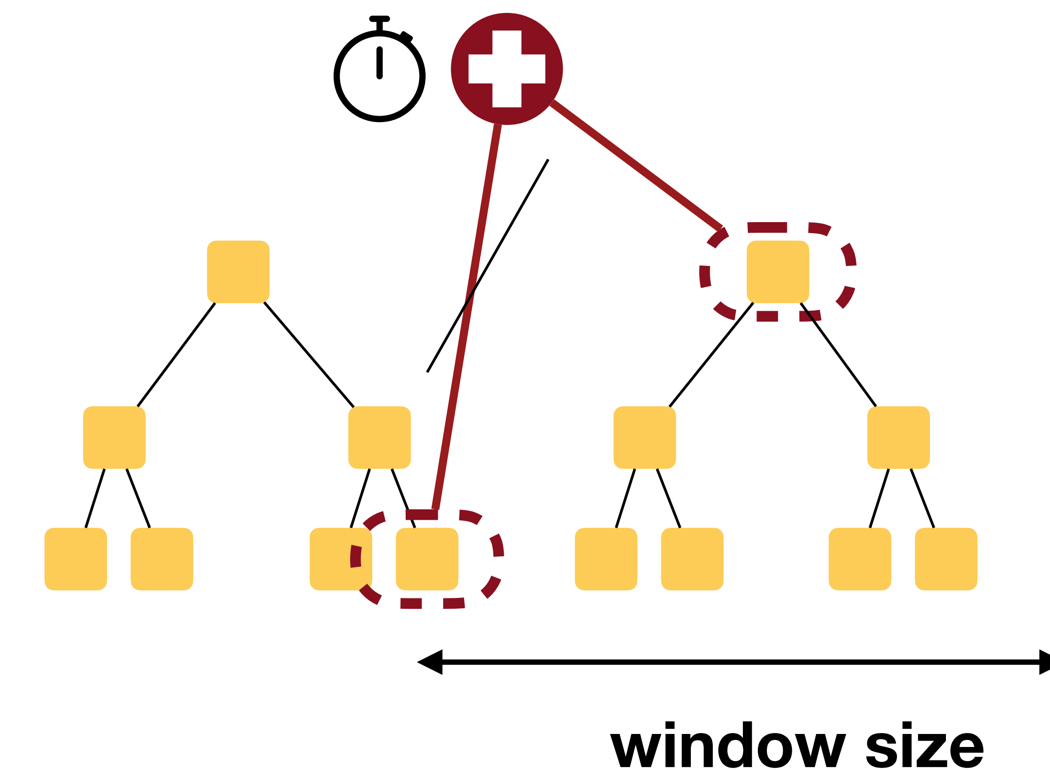
Use a binary tree to store partial aggregates on top of stream events.

On event: Insert to the binary tree and update all affected partial aggregates.

On trigger: Combine partial aggregated to compute and emit the final value.

The state requirements are high as both events and partial aggregates are maintained.

The evaluation function is applied on event and on trigger.



- Partial aggregates can be shared across overlapping windows.
- Memory copy for out-of-order events or tree re-balancing.
- Low-latency trigger as final aggregates can be computed by combining the pre-computed partial aggregates.

Window ID (Bucket)

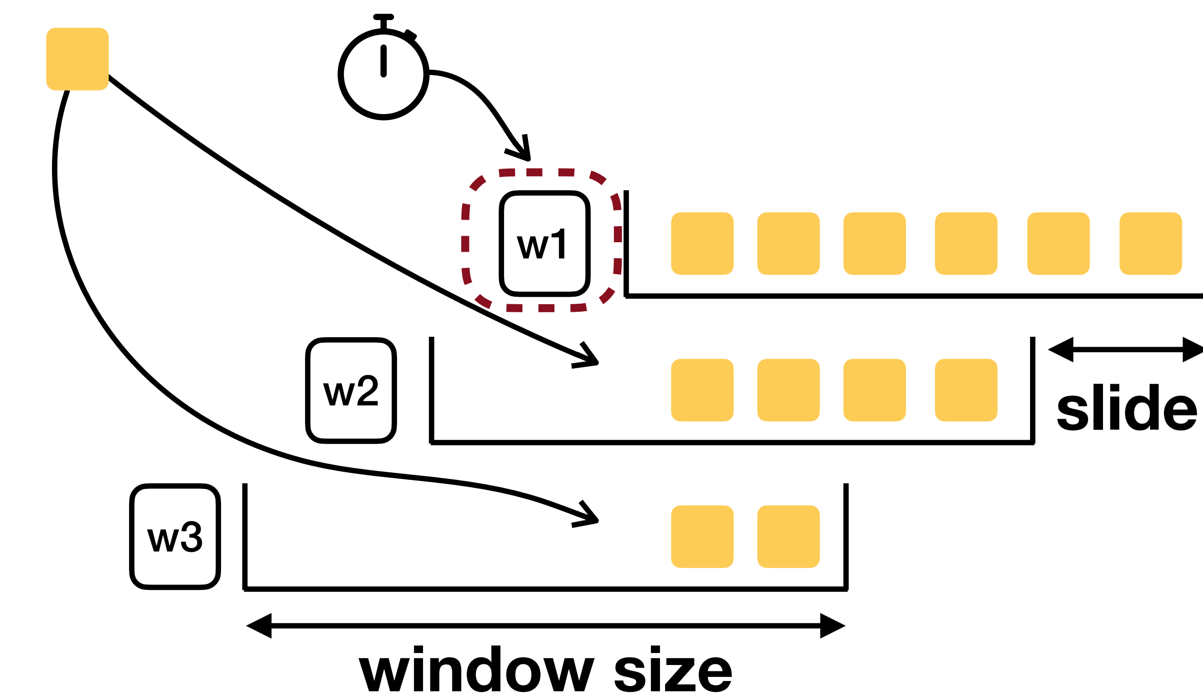
Organize events into windows by assigning them IDs (start or end timestamp).

On event: An assigner function computes a list of at most $\lceil \frac{l}{l_s} \rceil$ window IDs the record belongs to. The event is inserted to each window in the list.

On trigger: Window contents are retrieved using the ID.

This strategy has low state requirements when the evaluation function is associative and commutative and can be eagerly applied on record arrival.

If the window slide is much smaller than the window length, successive windows have large overlap, resulting to redundant computations and high state requirements.



Window ID (Bucket)

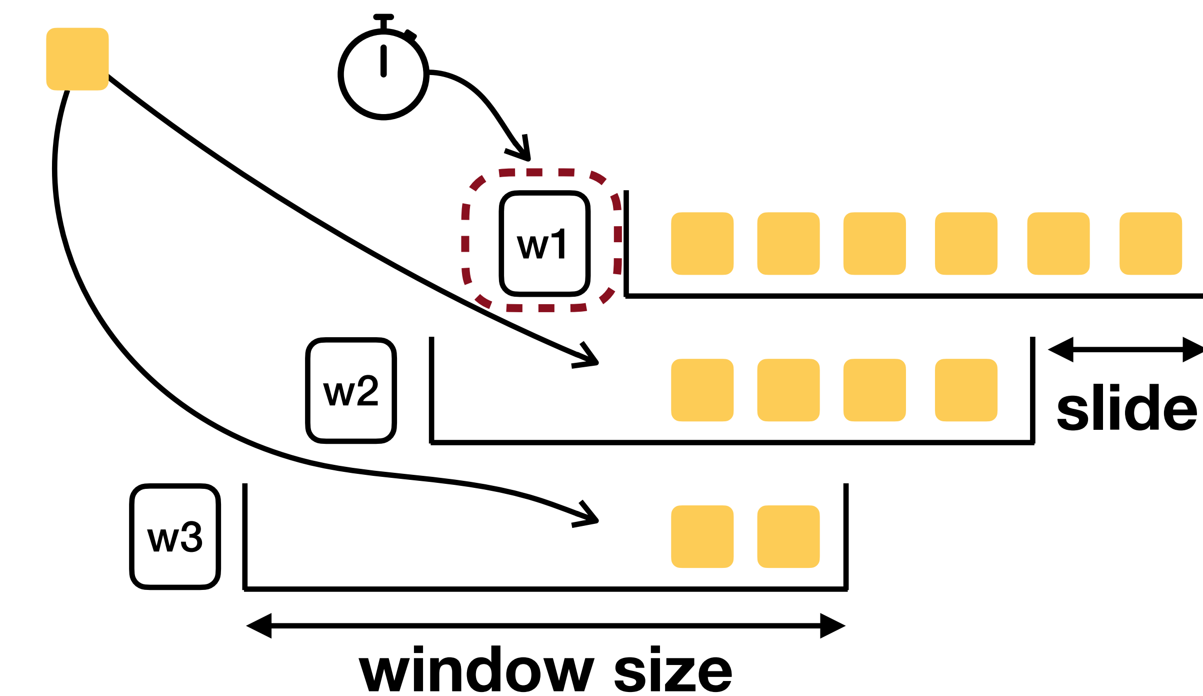
Organize events into windows by assigning them IDs (start or end timestamp).

On event: An assigner function computes a list of at most $\lceil \frac{l}{l_s} \rceil$ window IDs the record belongs to. The event is inserted to each window in the list.

On trigger: Window contents are retrieved using the ID.

This strategy has low state requirements when the evaluation function is associative and commutative and can be eagerly applied on record arrival.

If the window slide is much smaller than the window length, successive windows have large overlap, resulting to redundant computations and high state requirements.



- Low latency on trigger if aggregation can be computed eagerly.
- Redundancy, high memory requirements, and high latency on event (many assignments) for overlapping windows.

Window Slicing

Organize events into smaller units, called **panes**.

A pane is the maximum shareable unit across windows and its size is computed as $l_p = gcd(l, l_s)$.

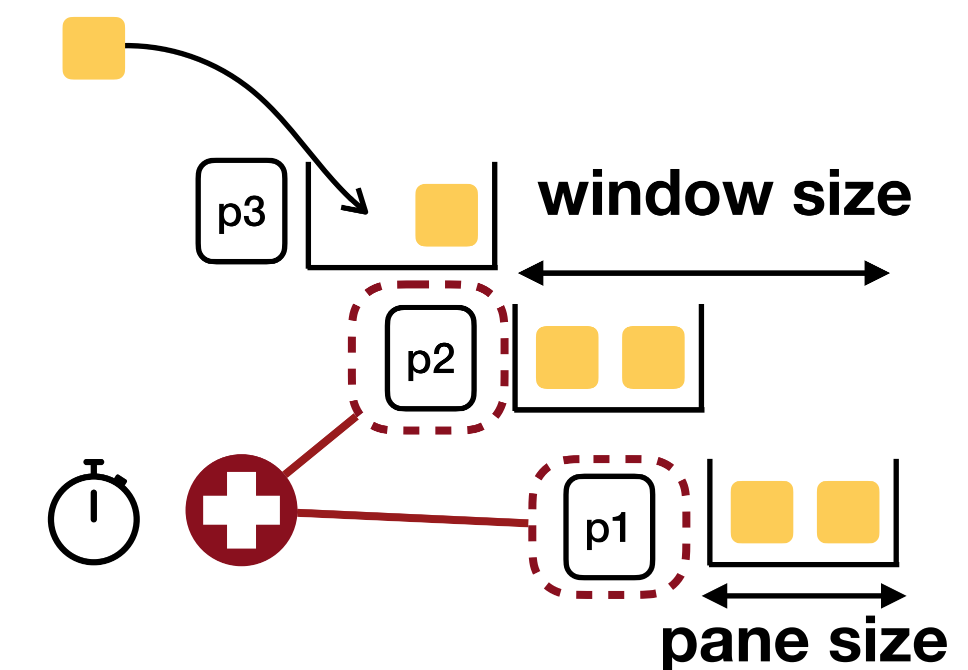
This guarantees that a record belongs to only one pane and that every window can be composed by a set of consecutive panes.

On event: An assigner function computes the pane ID and adds the record to its state.

On trigger: Retrieve $\frac{l}{l_p}$ panes to assemble the window contents.

If the evaluation function supports pre-aggregation, it can be eagerly applied on record arrival to maintain partially aggregated results per pane.

The final aggregate is computed by combining the partial aggregates on trigger.



🤧😷🤒 Vasiliki Kalavri | Boston University 2021

Window Slicing

Organize events into smaller units, called **panes**.

A pane is the maximum shareable unit across windows and its size is computed as $l_p = gcd(l, l_s)$.

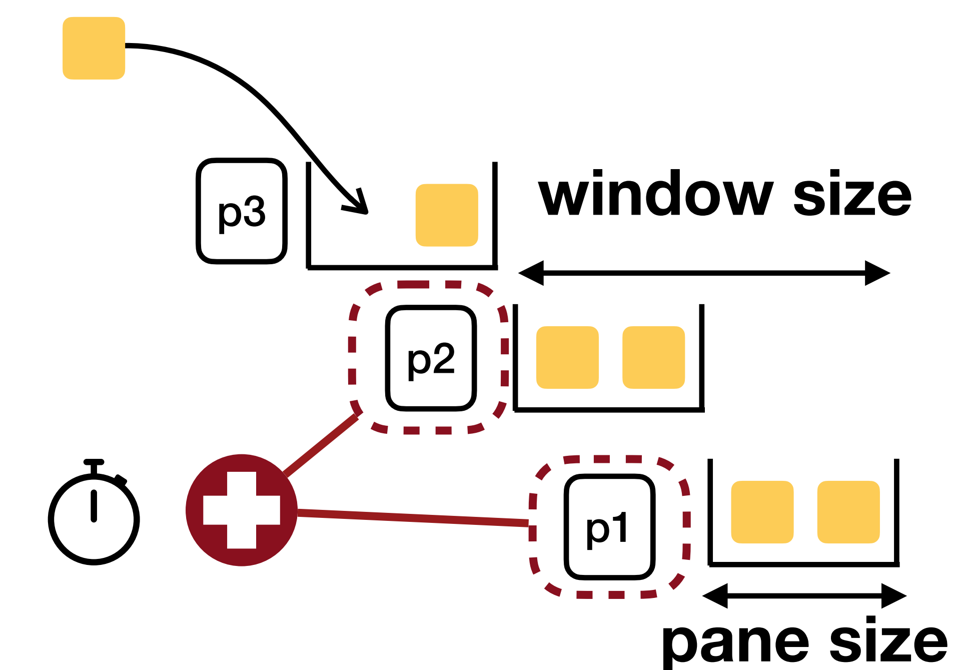
This guarantees that a record belongs to only one pane and that every window can be composed by a set of consecutive panes.

On event: An assigner function computes the pane ID and adds the record to its state.

On trigger: Retrieve $\frac{l}{l_p}$ panes to assemble the window contents.

If the evaluation function supports pre-aggregation, it can be eagerly applied on record arrival to maintain partially aggregated results per pane.

The final aggregate is computed by combining the partial aggregates on trigger.



- Low state requirements and low latency on trigger.
- When the window length is a multiple of the window slide, results can be shared across windows.

Flink window functions

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)))
```

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

```java
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

```
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)
    }
}
```

AggregateFunction example

```
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

AggregateFunction example

```
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

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)
    }
}
```

AggregateFunction example

```
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)
    }
}
```

AggregateFunction example

```
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();
    }
}
```

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

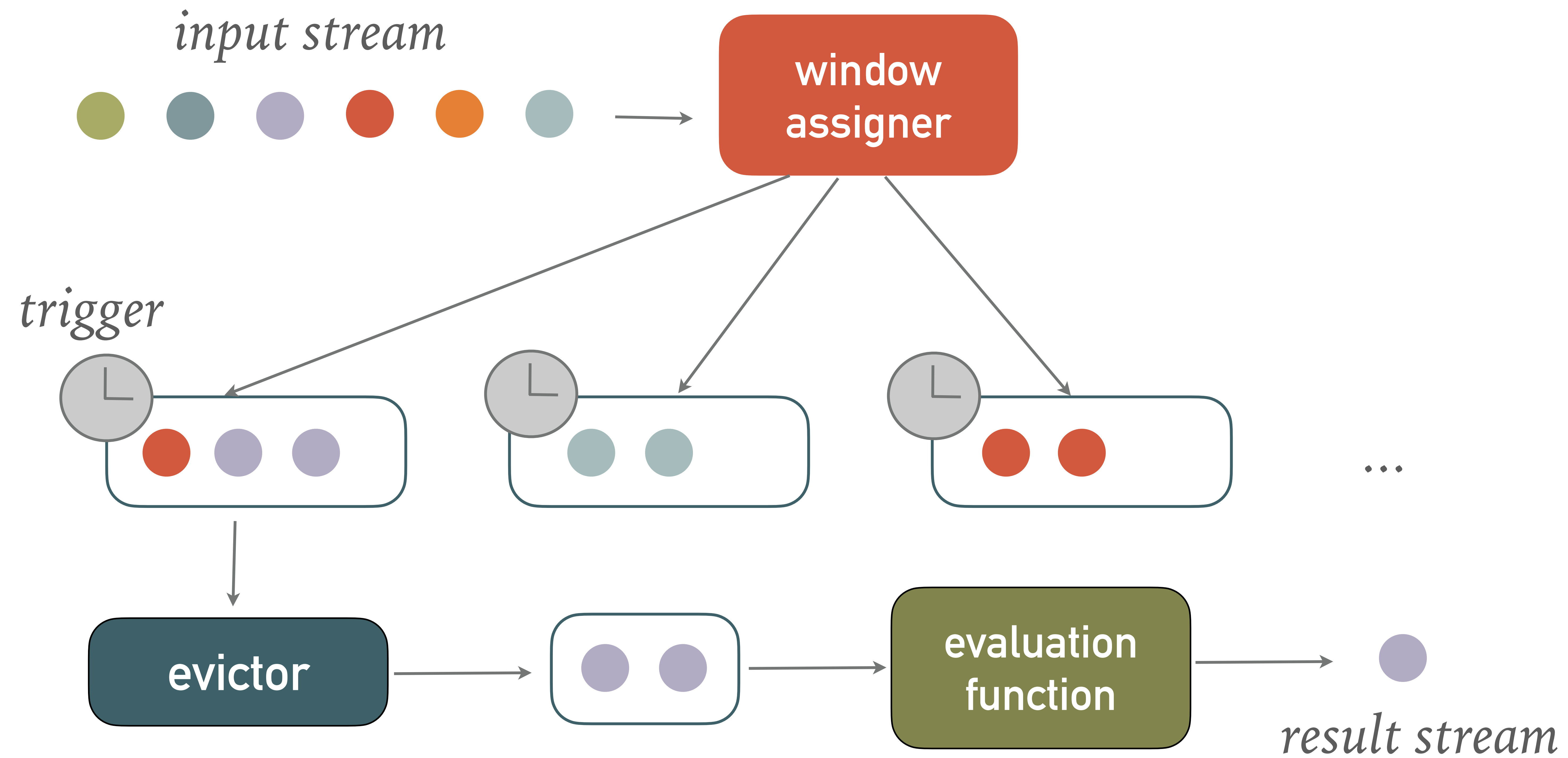
ProcessWindowFunction interface

```
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



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.

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)
```

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

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 = {…}
}
```

References

- Li, Jin, et al. "**Semantics and evaluation techniques for window aggregates in data streams**." *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005.
- Tangwongsan, Kanat, et al. "**General incremental sliding-window aggregation**." *Proceedings of the VLDB Endowment* 8.7 (2015): 702-713.
- Li, Jin, et al. "**No pane, no gain: efficient evaluation of sliding-window aggregates over data streams**." *Acm Sigmod Record* 34.1 (2005): 39-44.
- Traub, Jonas, et al. "**Efficient Window Aggregation with General Stream Slicing**." *EDBT*. 2019.