CS 591 K1:
Data Stream Processing and Analytics

Spring 2021

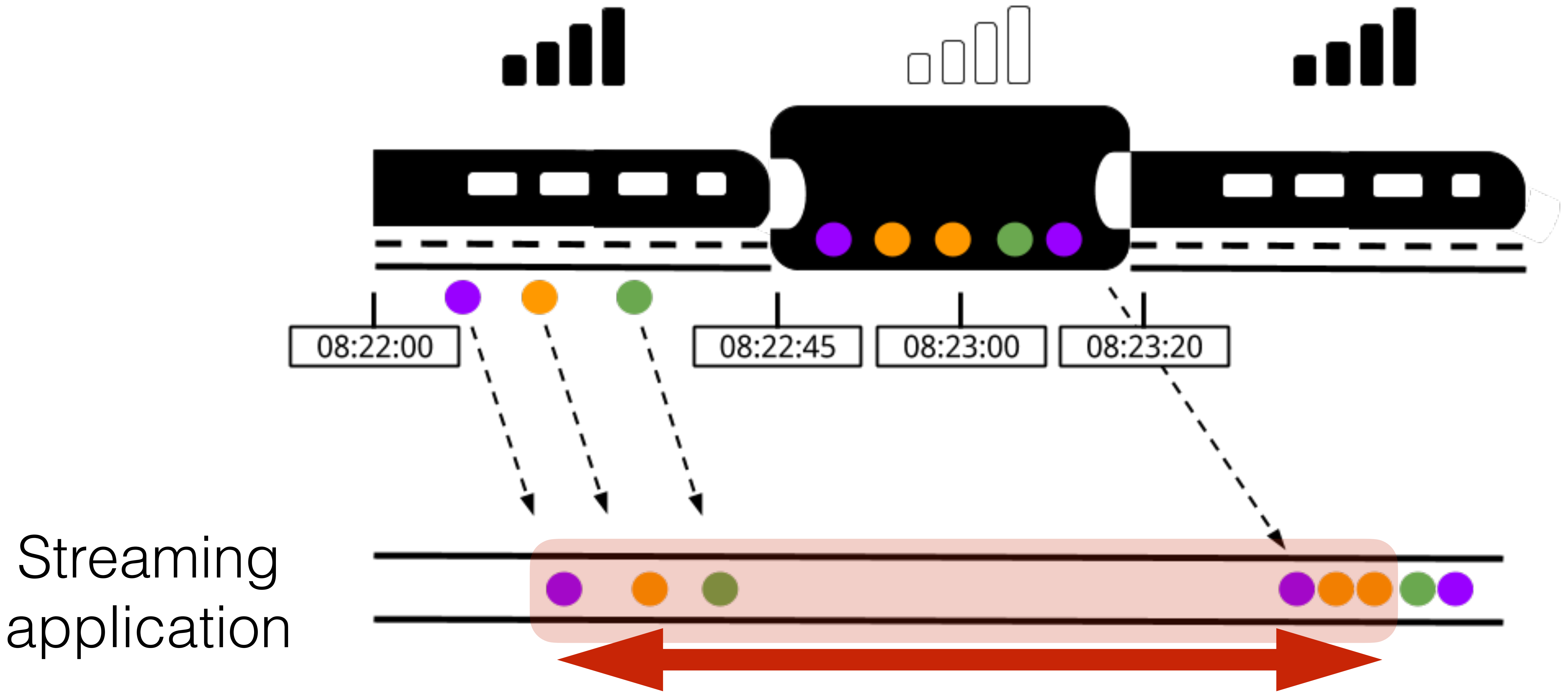
Notions of time and progress

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Topics covered in this lecture

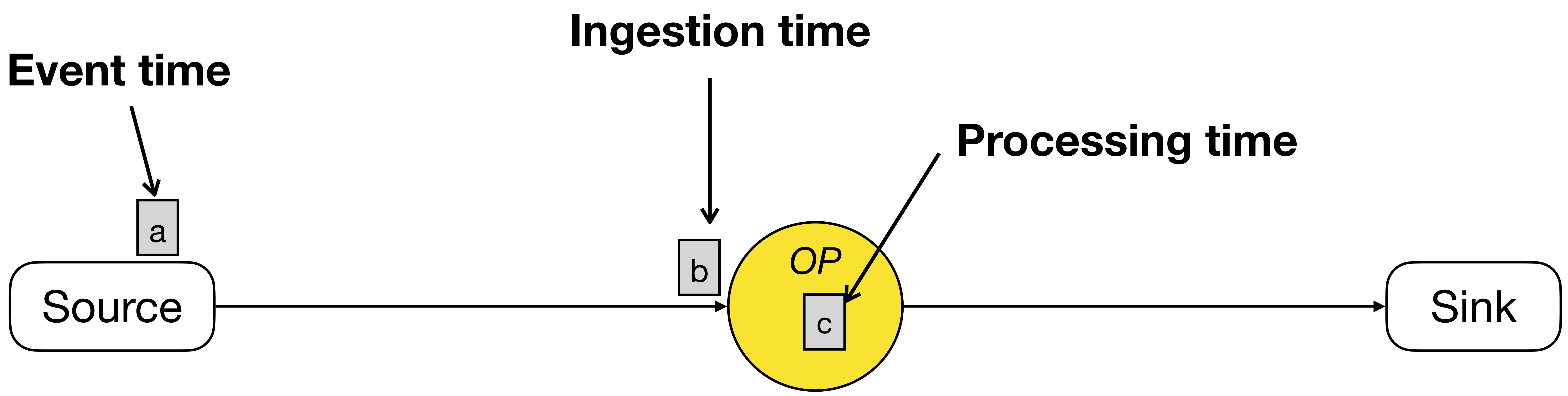
- What's the meaning of one minute?
  - different notions of time
  - application time skew
- Watermarks
  - propagation, trade-offs
  - late data handling
- Heartbeats
  - automatic generation
  - guarantees, ensuring progress

What's the meaning of one minute?



Notions of time

- **Event** time
  is the time tuples are generated at the sources. Also called **application** time.
- **Processing** time
  is the time tuples are processed in a streaming system.
- **Ingestion** time
  is the time tuples arrive in a streaming system.



This is called **event time**

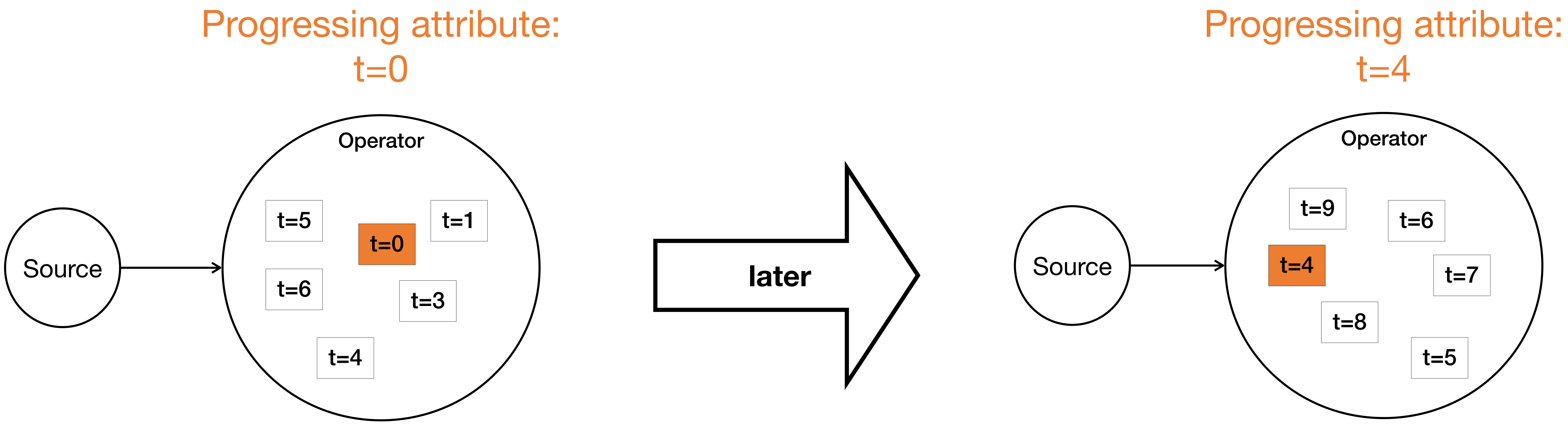
Episode **IV**: *A New Hope*	Episode **V**: *The Empire Strikes Back*	Episode **VI**: *Return of the Jedi*	Episode **I**: *The Phantom Menace*	Episode **II**: *Attach of the Clones*	Episode **III**: *Revenge of the Sith*	Episode **VII**: *The Force Awakens*
1977	*1980*	*1983*	*1999*	*2002*	*2005*	*2015*

This is called ***processing time***

Progress

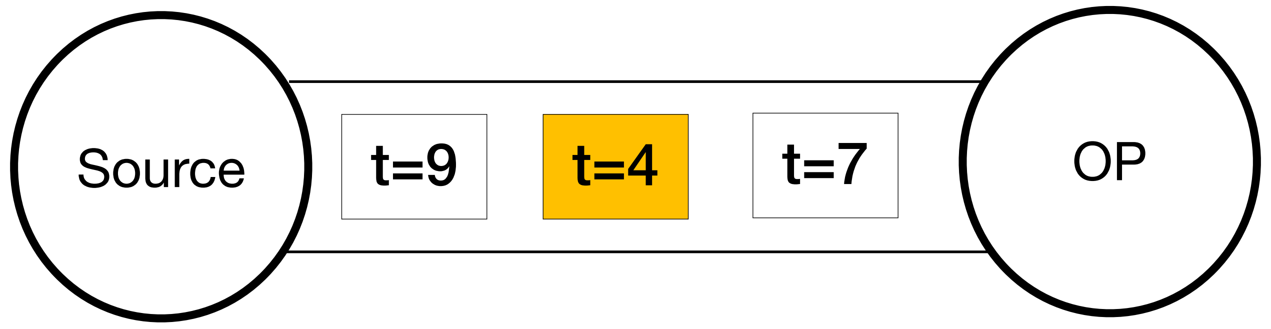
Assuming that a stream is ordered by one of its attributes *A* in increasing order, then the processing of the stream progresses when **the smallest value of A** among the unprocessed tuples increases over time.

A is called a *progressing attribute*, e.g. the event time timestamp.



Out-of-order data

Out-of-order data tuples arrive in a streaming system after tuples with later event time timestamps.



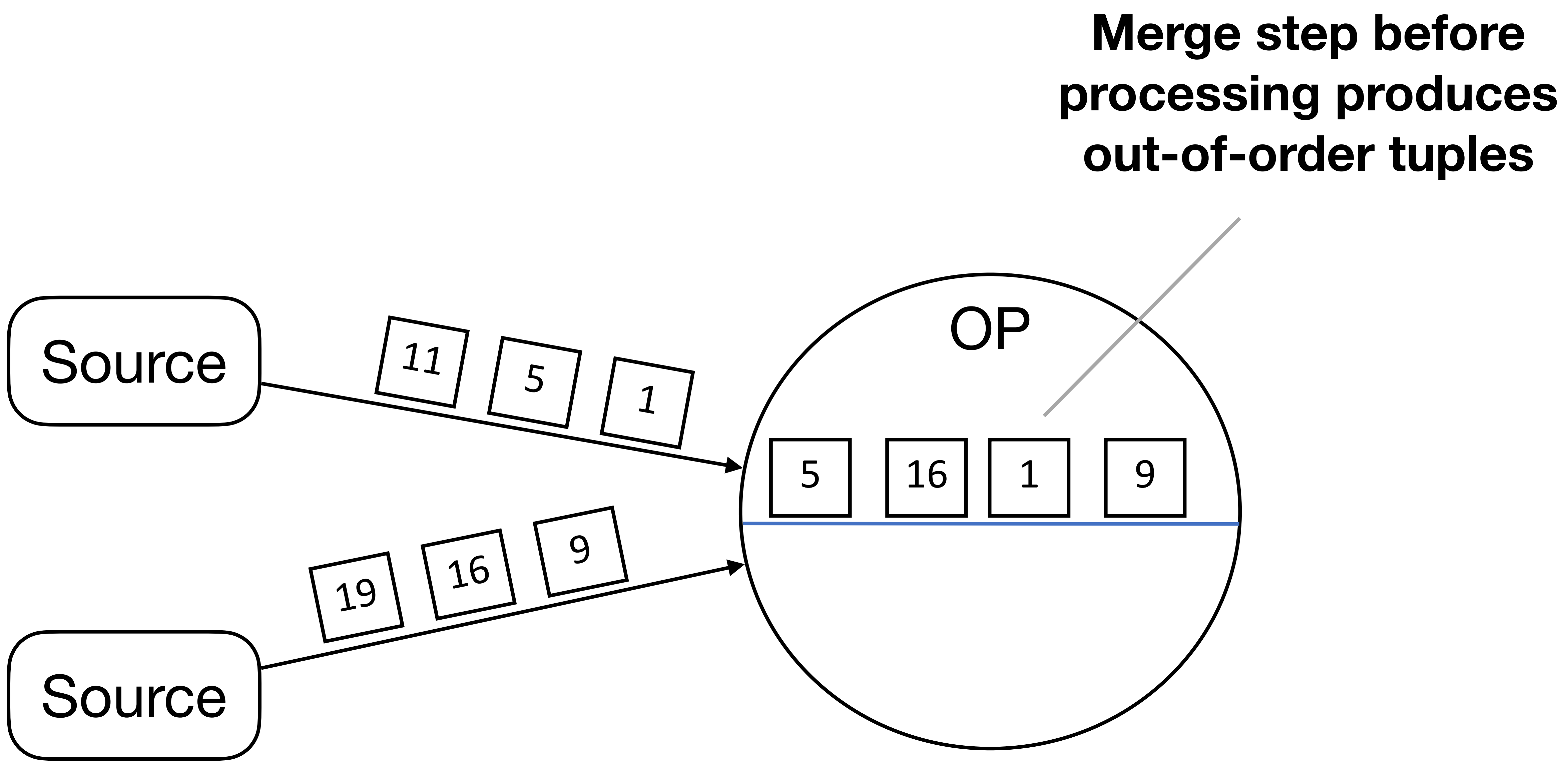
Causes of disorder

- External stochastic factors
- System operations

Causes of disorder

External stochastic factors

- Network routing
- Multiple input sources

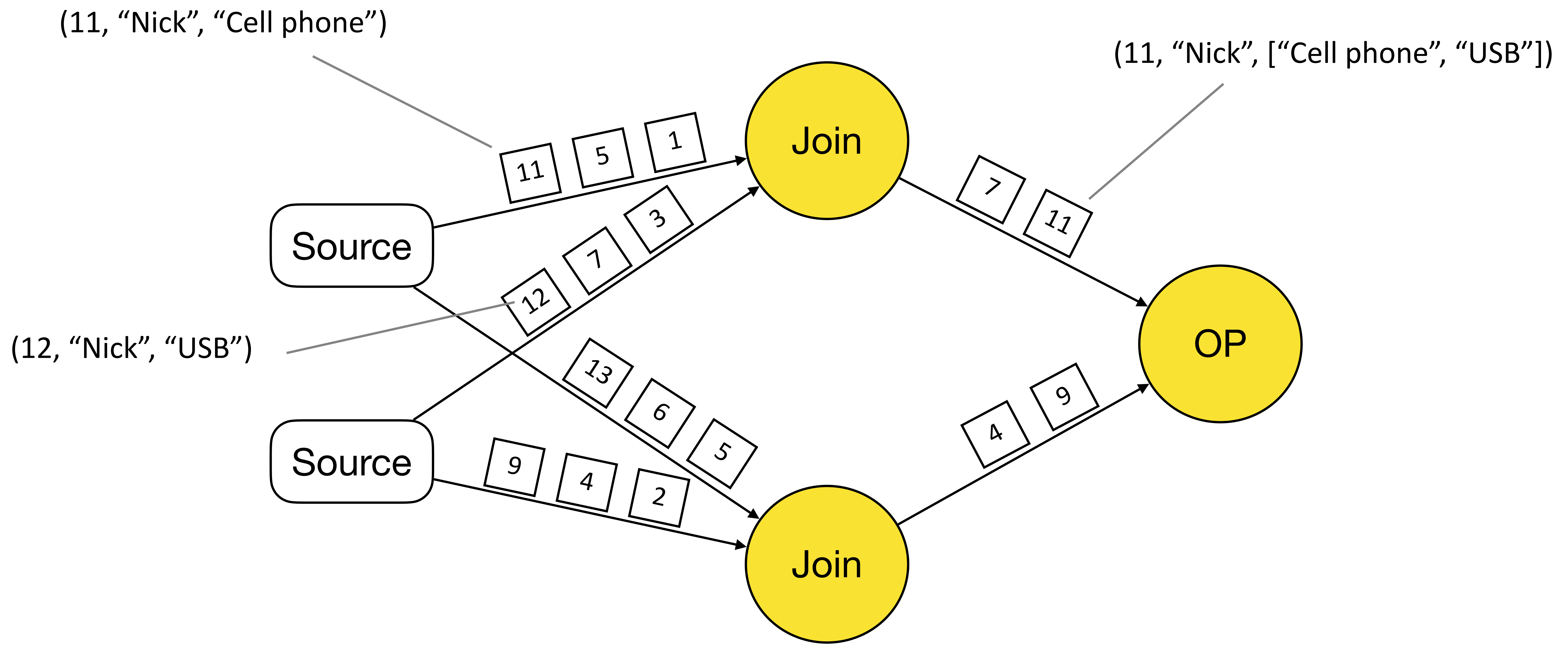


Causes of disorder

System operations

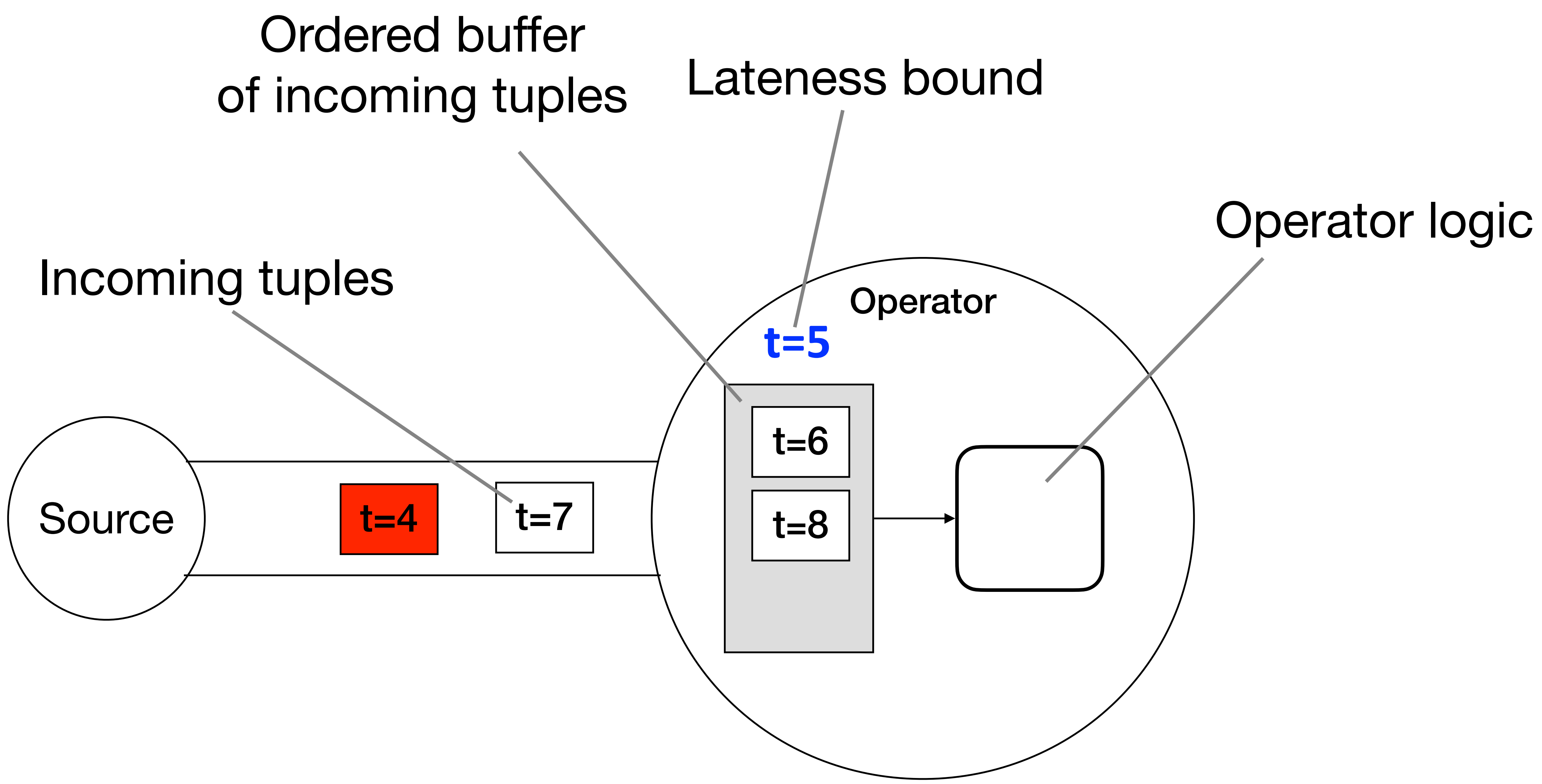
- A parallel join operator produces a shuffled combination of the two joined streams and output results in the order of match.
- A union operator on two unsynchronized streams yields a stream with all tuples of the two input streams interleaving each other in random order.
- Windowing based on an attribute that is different to the ordering attribute reorders the stream.
- Data prioritization using an attribute different to the ordering one changes the order.

Disorder caused by system operation



In-order architecture

- Buffer incoming tuples
- Reorder incoming tuples
- Push tuples to the operator logic according to a *lateness* bound and ignore tuples that arrive to the operator after that.



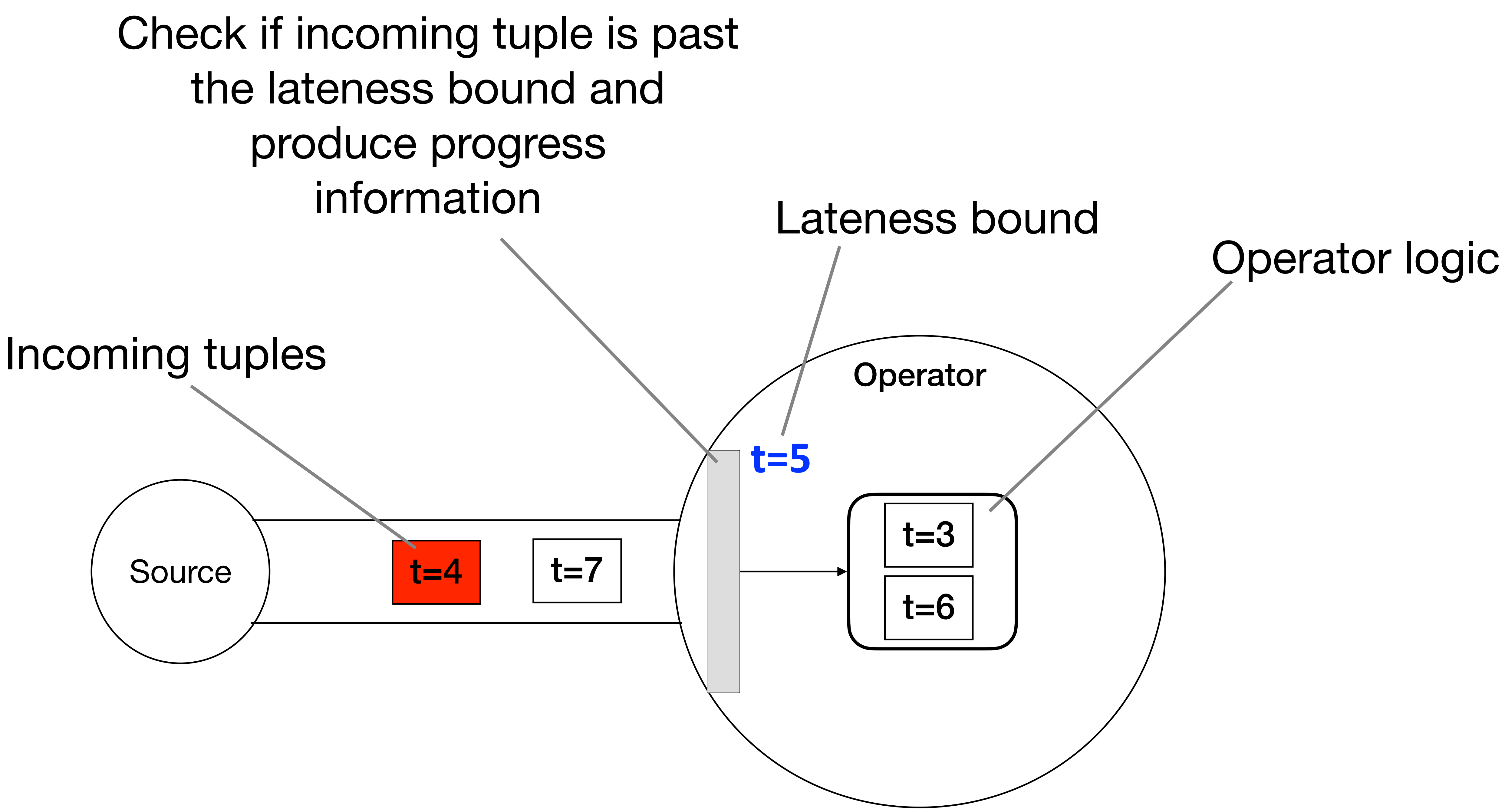
Out-of-order architecture

Out-of-order architecture

- Operators or a global authority produce **progress** information using some metric and propagate it to the dataflow graph.
- The progress information typically reflects the oldest unprocessed tuple in the system and establishes a lateness bound for admitting out-of-order tuples.
- In contrast to in-order systems, tuples are processed **in the order of their arrival** up to the lateness bound.

Out-of-order architecture

- Admit incoming tuples that are not past the lateness bound and ignore the rest
- The lateness bound typically reflects the oldest pending work
- Update progress information
- Propagate progress information to the data flow graph



Effects of disorder

Leads to wrong results if ignored

- Dropping a tuple that arrived after its time will make a join computation incorrect

Impedes processing progress for order-sensitive operators (join, aggregate)

- In-order architecture systems
  - Buffer and reorder data as they come
  - Add processing overhead, memory space overhead, and latency

Out-of-order architecture systems

  - Establish bound based on processing progress and process tuples since that point without reordering
  - Stock processing state
  - Add implementation complexity

Except for order-agnostic operators

- project, filter, map, union

Progress-tracking mechanisms

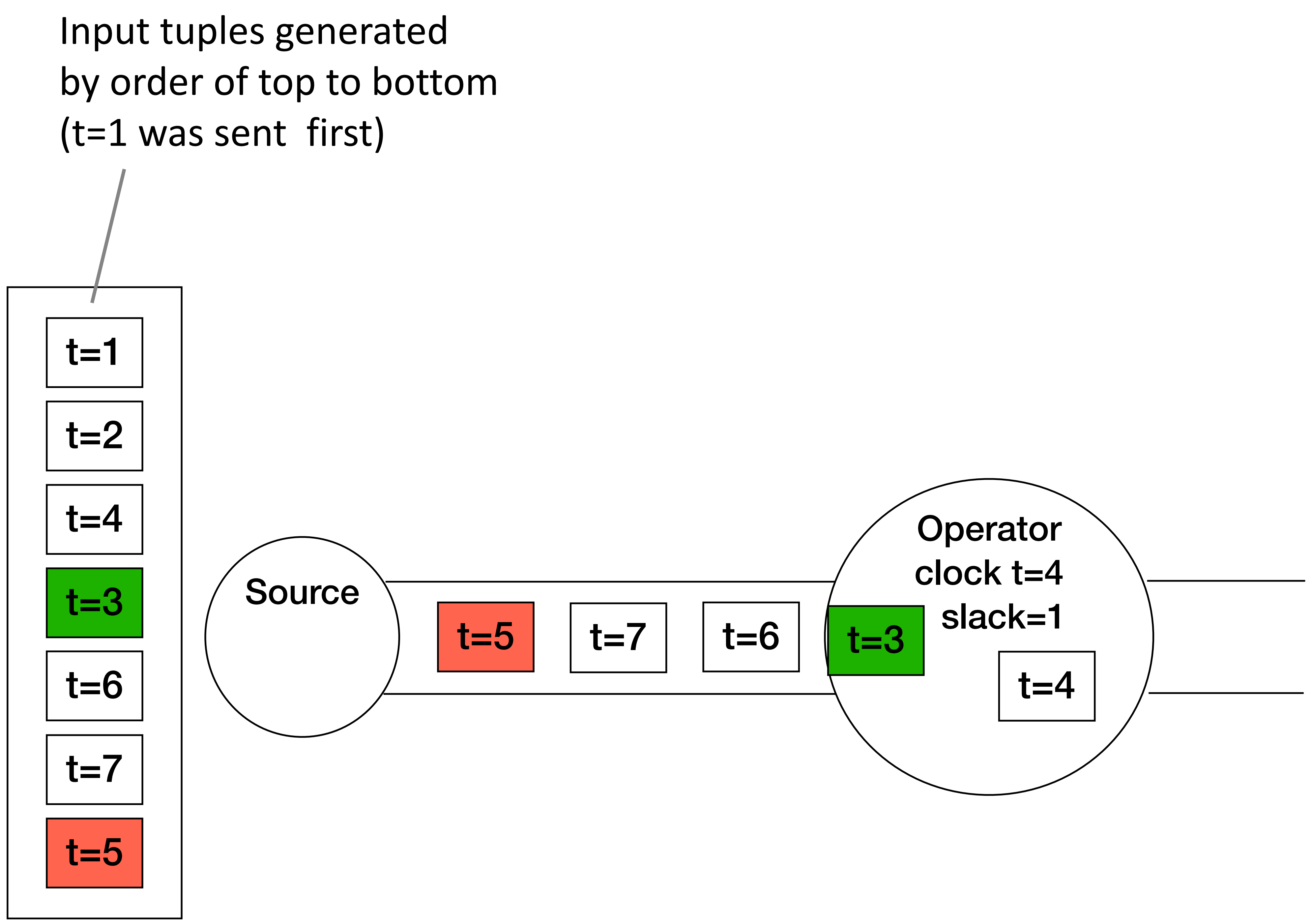
- Slack
- Heartbeat
- Low-watermark
- Pointstamp and frontier, see Naiad SOSP'13

Slack

- Wait for out-of-order data for a fixed amount of a certain metric.
- Originally denoted the number of tuples intervening between the actual occurrence of an out-of-order tuple and the position it would have in the input stream if it arrived on time.
- Can also be quantified in terms of time.
- Slack marks a fixed **grace period** for late tuples.

Slack in action

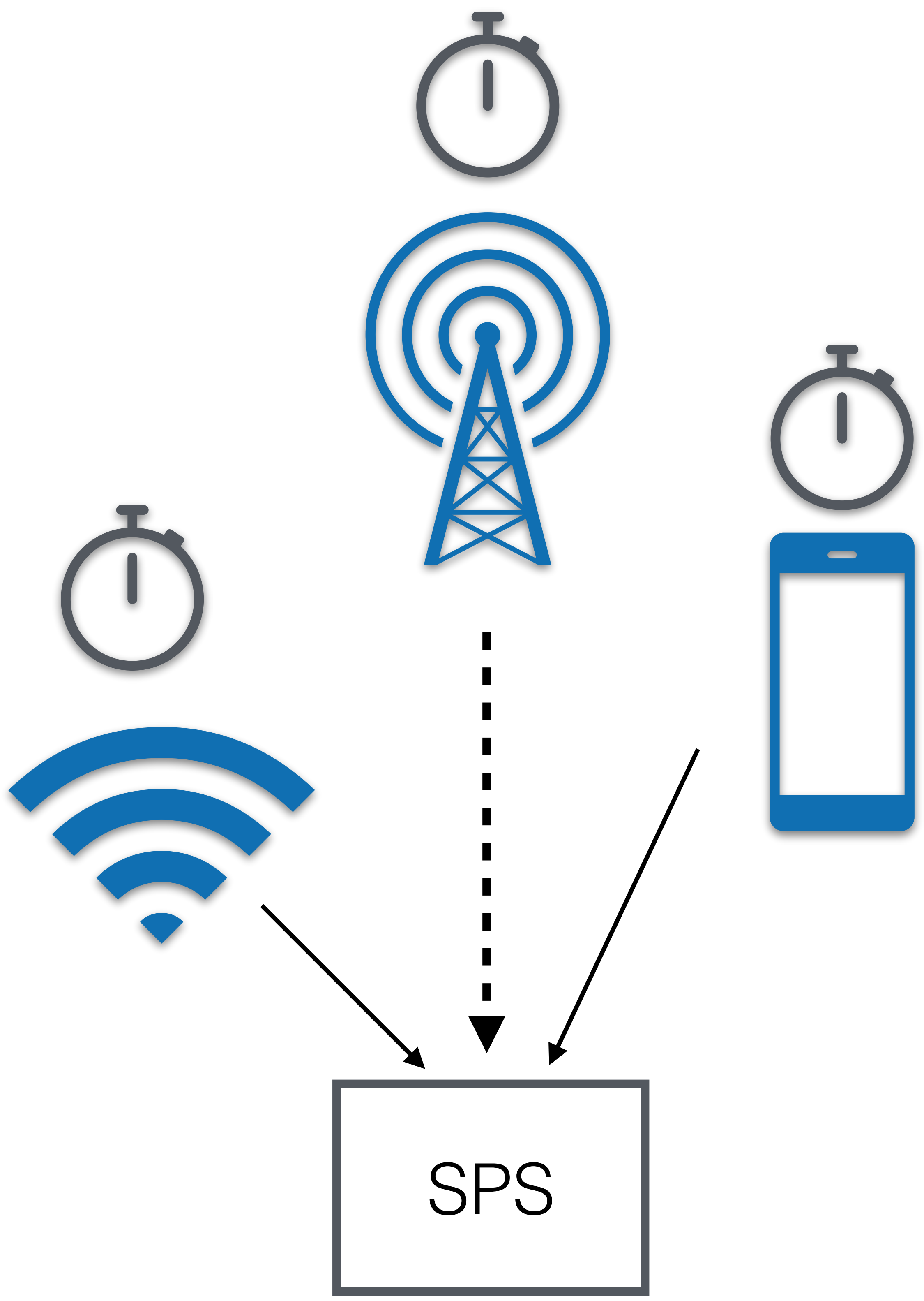
- Close first window [0,4) when t=3 arrives
- Normally window would close when t=4 arrives, but because of slack=1 window closing awaits the next tuple that will make the slack expire
- Because t=3 arrives it is included in the window
- The window will output C=3 for t=1, t=2, and t=3
- Admit t=3 because of slack=1

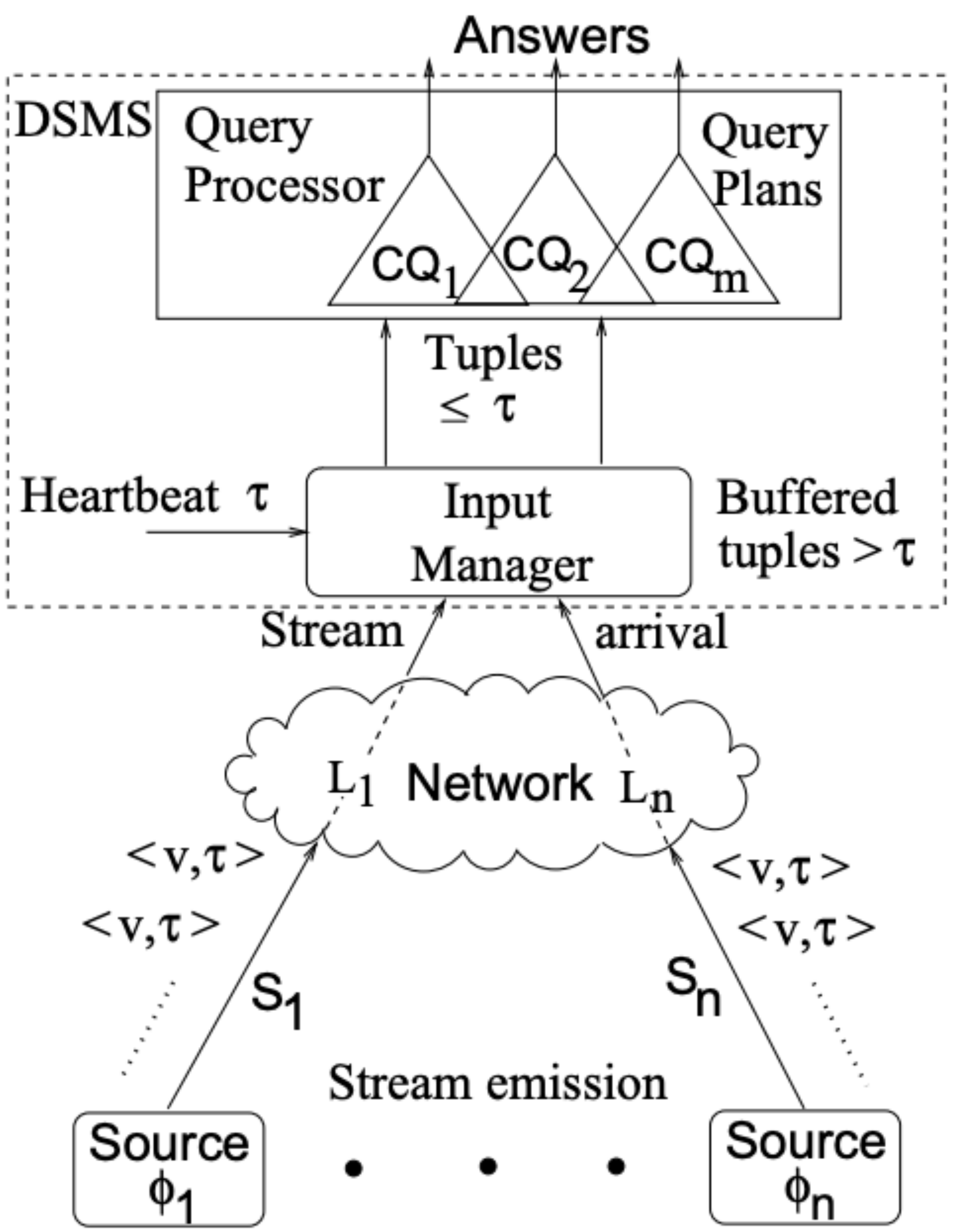


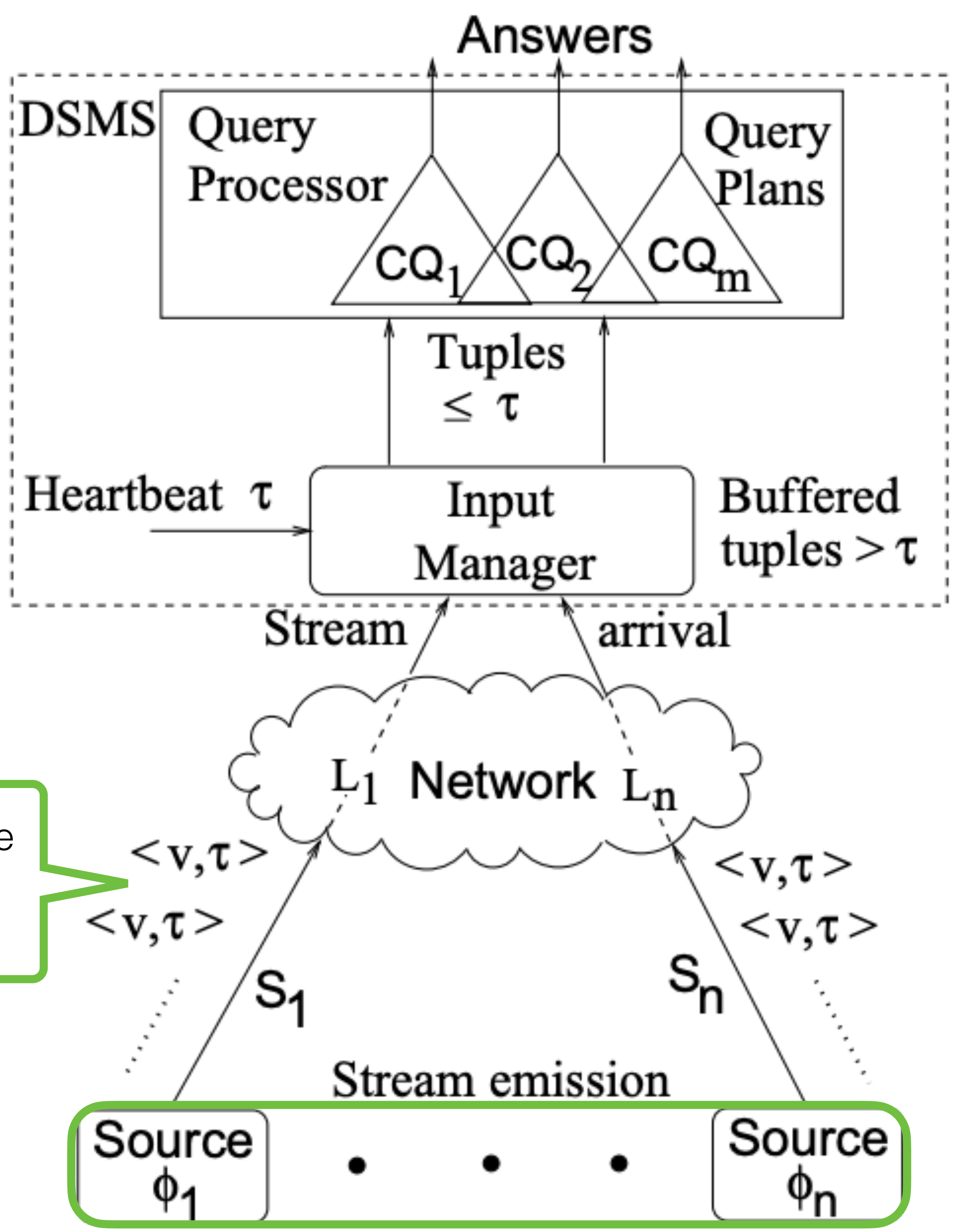
Heartbeats

Causes of event-time skew

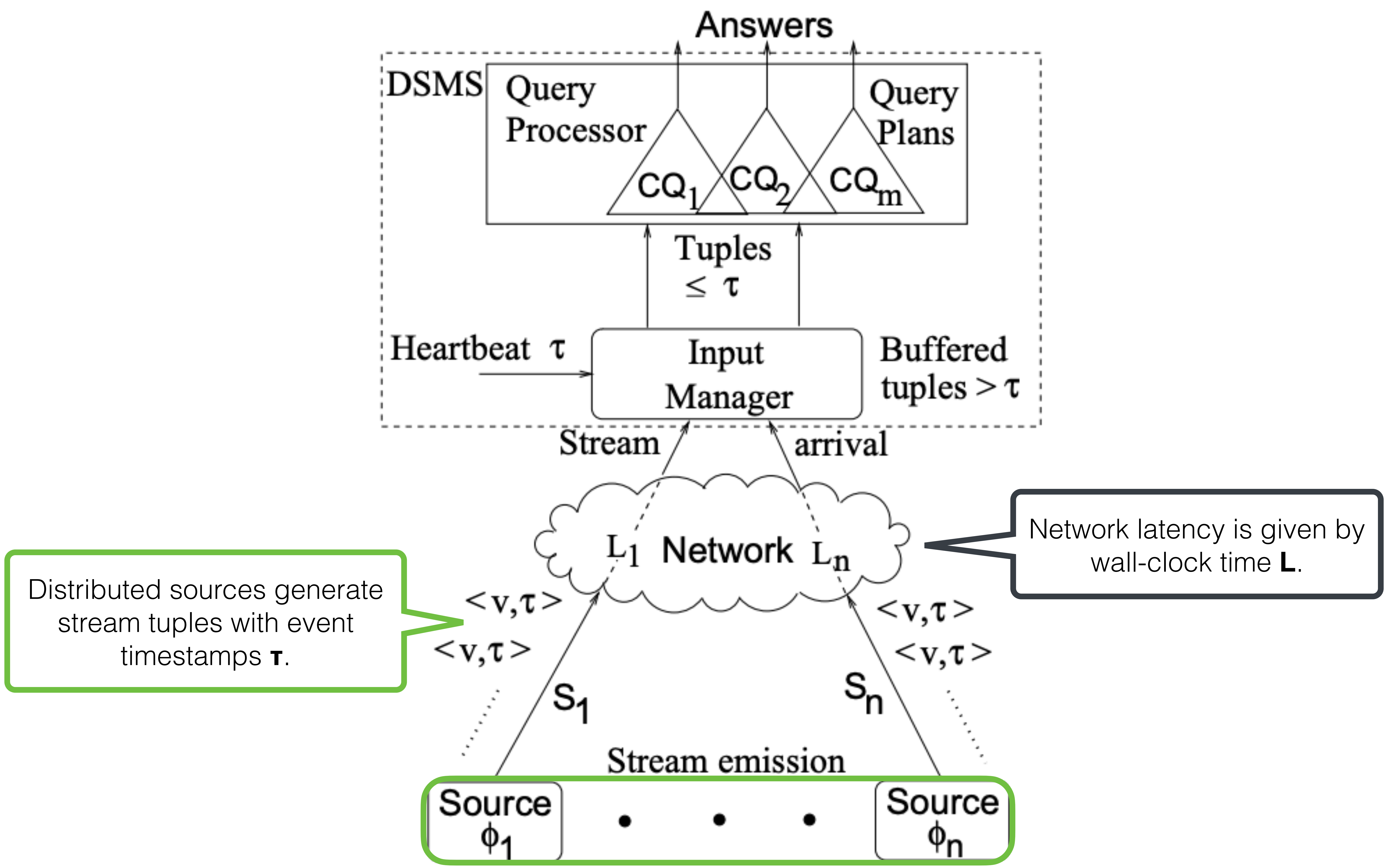
- Unsynchronized clocks at the sources
- Different latencies from different sources to the system
- Data transmission over a non-order-preserving channel




Answers
DSMS
Query Processor
Query Plans
CQ_1
CQ_2
CQ_m
Tuples $\leq \tau$
Heartbeat τ
Input Manager
Buffered tuples $> \tau$
Stream
arrival
L_1
Network
L_n
<v,τ>
<v,τ>
S_1
S_n
<v,τ>
<v,τ>
Stream emission
Source ϕ_1
Source ϕ_n

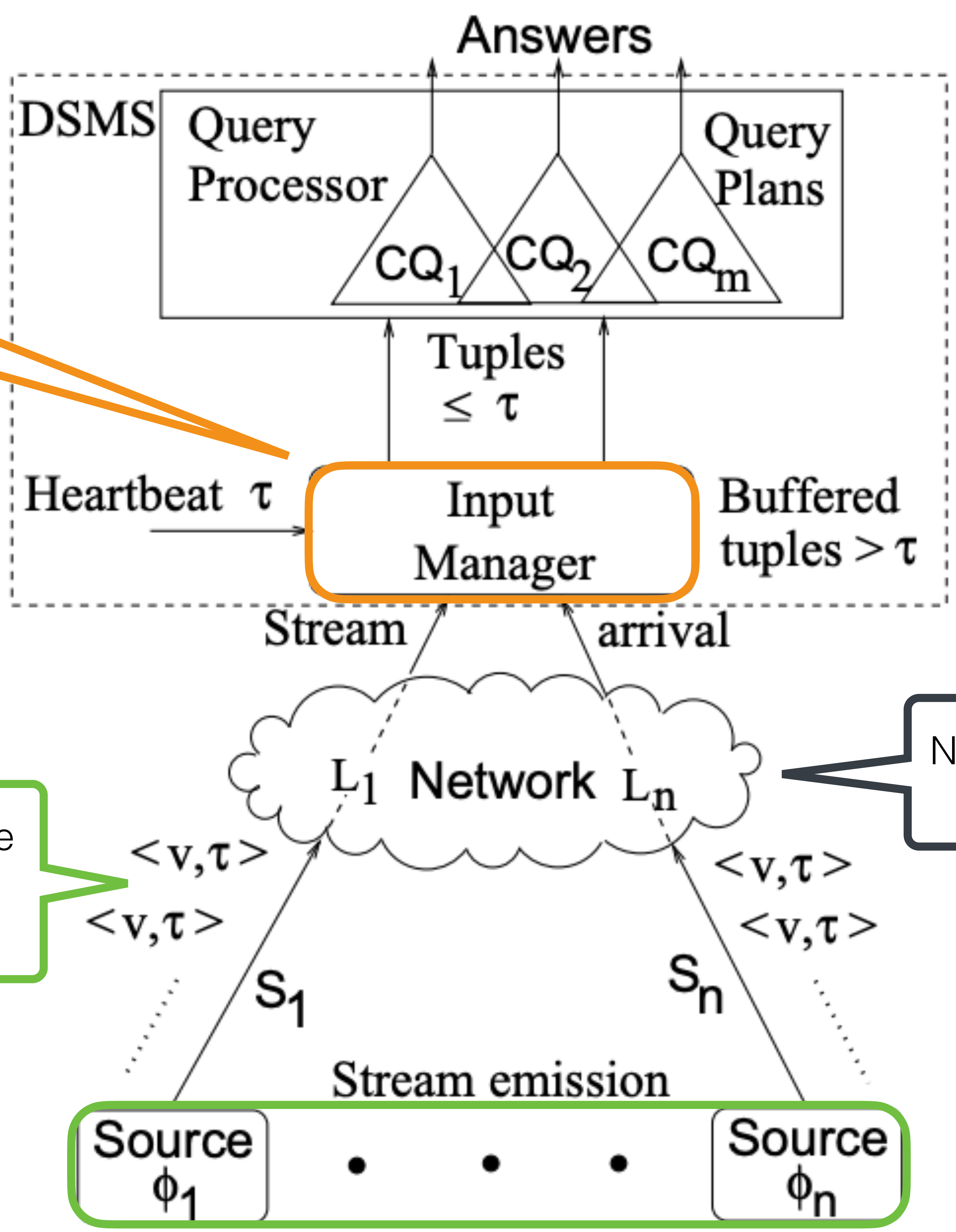


Distributed sources generate stream tuples with event timestamps **τ**.



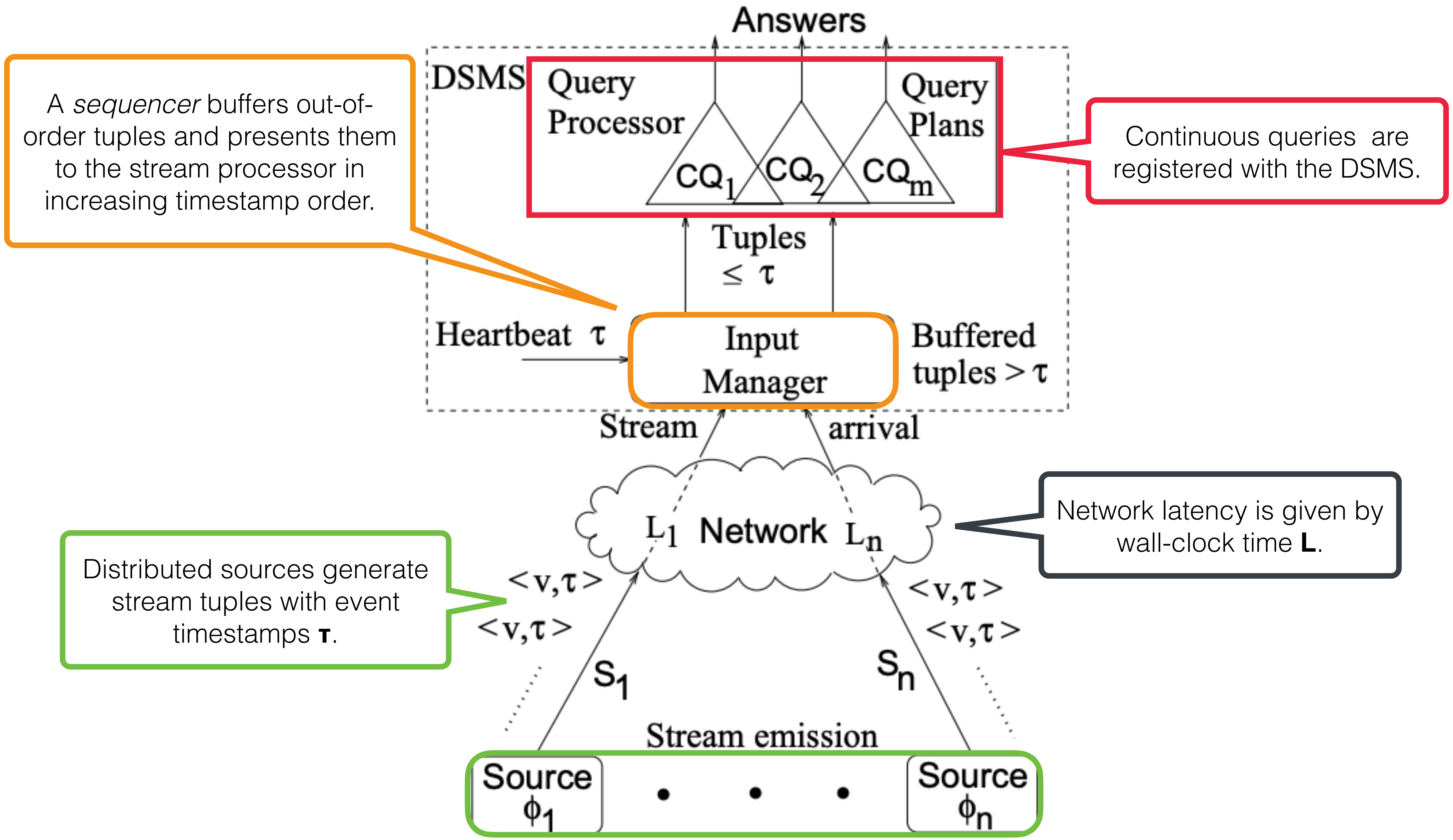
Answers
DSMS
Query Processor
Query Plans
CQ_1
CQ_2
CQ_m
Tuples $\leq \tau$
Heartbeat τ
Input Manager
Buffered tuples $> \tau$
Stream arrival
L_1 Network L_n
$<v,\tau>$
$<v,\tau>$
$<v,\tau>$
$<v,\tau>$
S_1
S_n
Stream emission
Source ϕ_1
Source ϕ_n
Distributed sources generate stream tuples with event timestamps **τ**.
Network latency is given by wall-clock time **L**.

A *sequencer* buffers out-of-order tuples and presents them to the stream processor in increasing timestamp order.

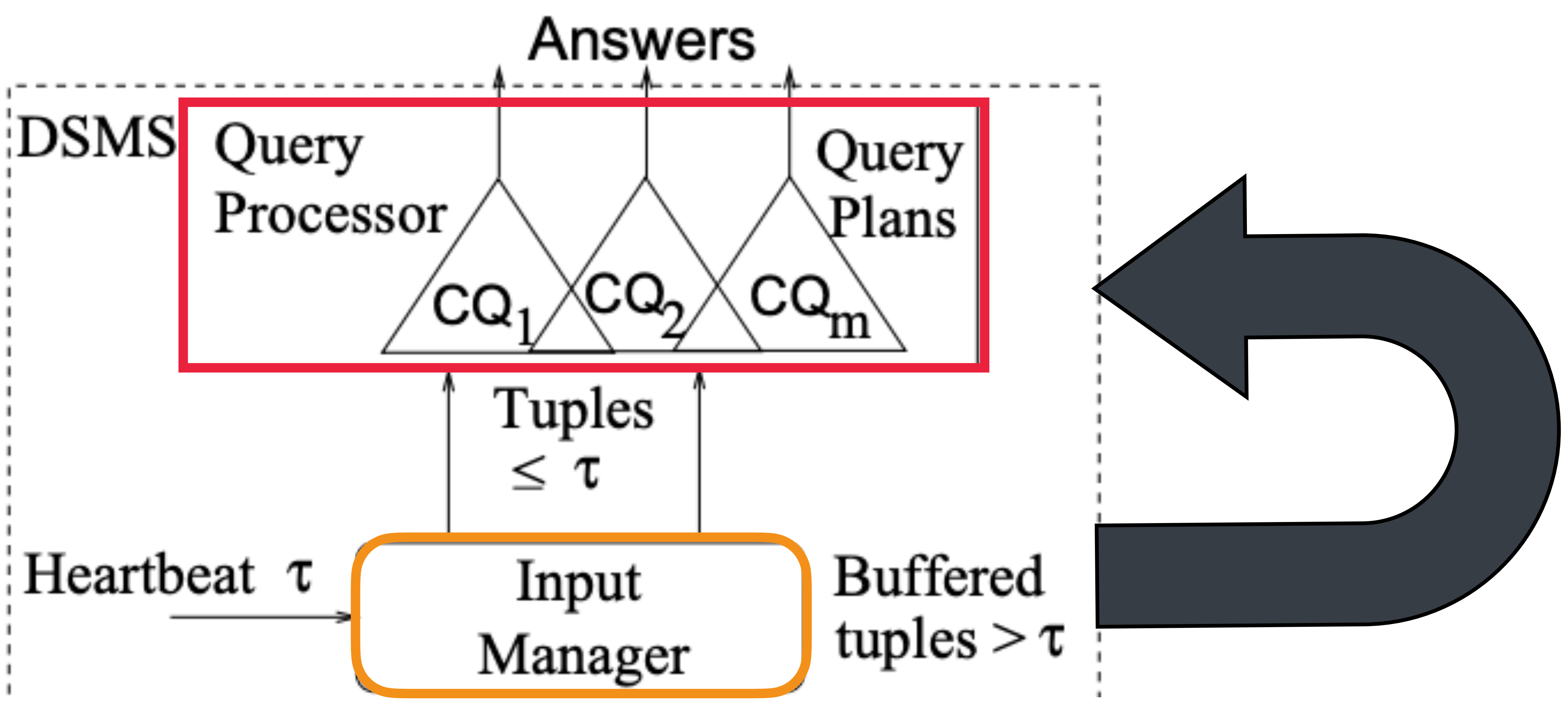


Network latency is given by wall-clock time **L**.

Distributed sources generate stream tuples with event timestamps **τ**.


A *sequencer* buffers out-of-order tuples and presents them to the stream processor in increasing timestamp order.
Answers
DSMS
Query Processor
Query Plans
CQ_1 CQ_2 CQ_m
Continuous queries are registered with the DSMS.
Tuples $\leq \tau$
Heartbeat τ
Input Manager
Buffered tuples $> \tau$
Stream arrival
L_1 Network L_n
Network latency is given by wall-clock time **L**.
Distributed sources generate stream tuples with event timestamps **τ**.
$<v,\tau>$
$<v,\tau>$
$<v,\tau>$
$<v,\tau>$
S_1
S_n
Stream emission
Source ϕ_1
Source ϕ_n

Progress requirement: Every stream element must *eventually* be moved from the input manager to the query processor without violating the ordering requirement.



Perfect Heartbeats

A heartbeat for a set of streams $S_1, S_2, ..., S_n$, at wall-clock time c is the maximum event timestamp τ so that all elements arriving on $S_1, S_2, ..., S_n$ after time c must have timestamp $> \tau$.

- If the above definition holds always and no late data ever arrive, the heartbeats are **perfect**.
- To deduce perfect heartbeats, there need to exist **bounds** on:
  1. event clock **skew** at the sources
  2. **out-of-order** generation of stream elements
  3. network **latency**

Skew bound

Given sources ϕ_i, ϕ_j

the pair (t_{ij}, δ_{ij}), $t_{ij} \geq 0, \delta_{ij} \geq 0$ denotes that:

if at time c, ϕ_i emits a tuple with timestamp τ

then all tuples emitted by ϕ_j after time $c + t_{ij}$

shall have timestamp $> \tau - \delta_{ij}$

Skew bound

Given sources ϕ_i, ϕ_j

the pair (t_{ij}, δ_{ij}), $t_{ij} \geq 0, \delta_{ij} \geq 0$ denotes that:

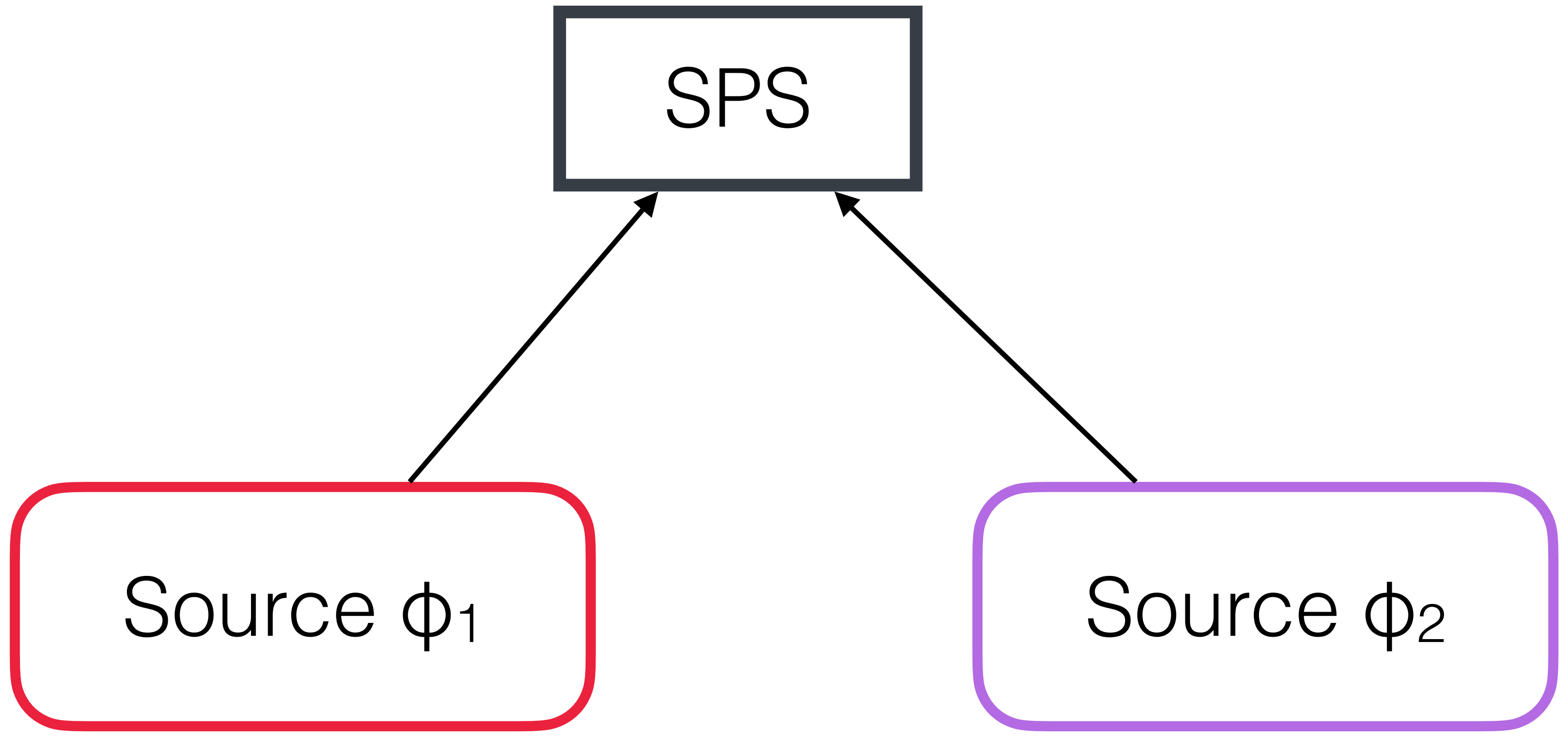
ϕ_j *lags behind* ϕ_i

by at most δ_{ij} units of event time

and this guarantee is delayed by

t_{ij} units of processing time.

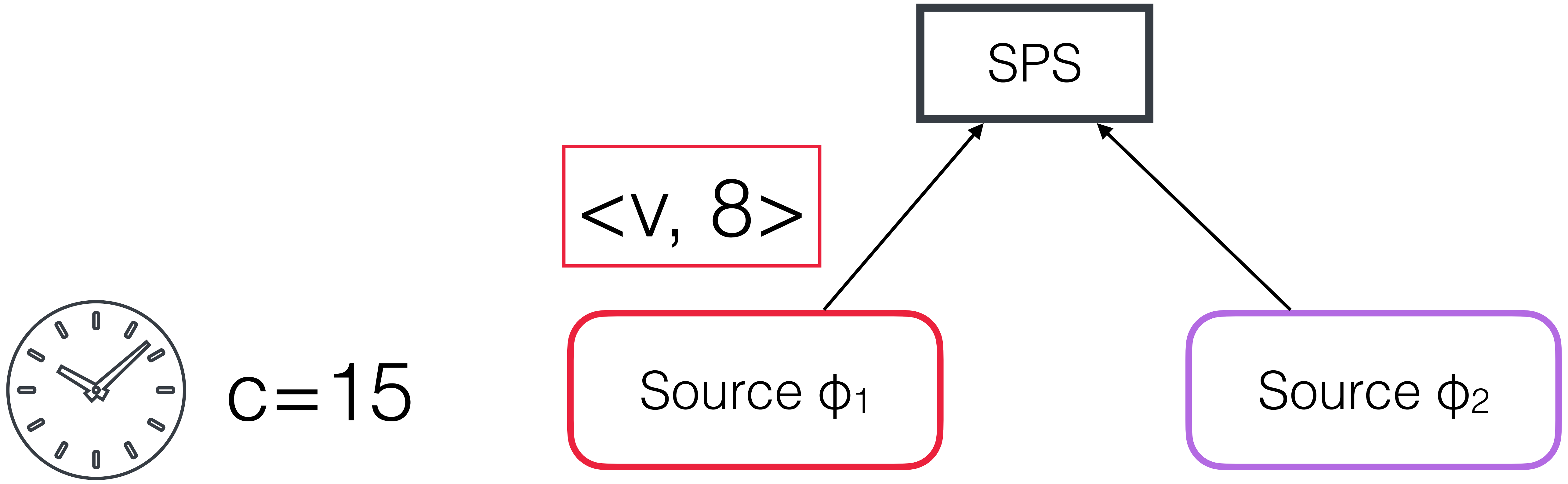
Skew bound: example



$$(t_{12}, \delta_{12}) = (2, 3)$$

Any tuple that ϕ_2 emits after c'=15 + 2 = 17
will have τ' > τ - δ = 8 - 3 = 5

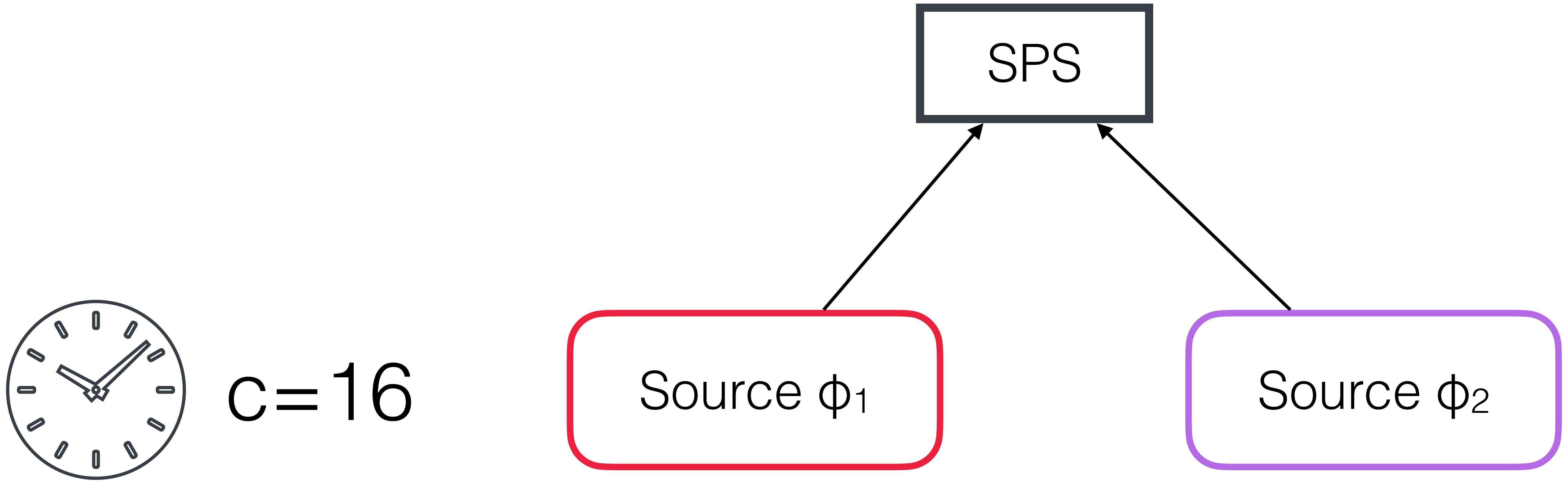
Skew bound: example



$$(t_{12}, \delta_{12}) = (2, 3)$$

Any tuple that ϕ2 emits after c'=15 + 2 = 17
will have τ' > τ - δ = 8 - 3 = 5

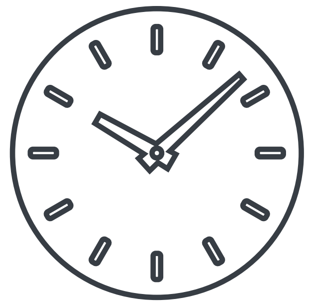
Skew bound: example



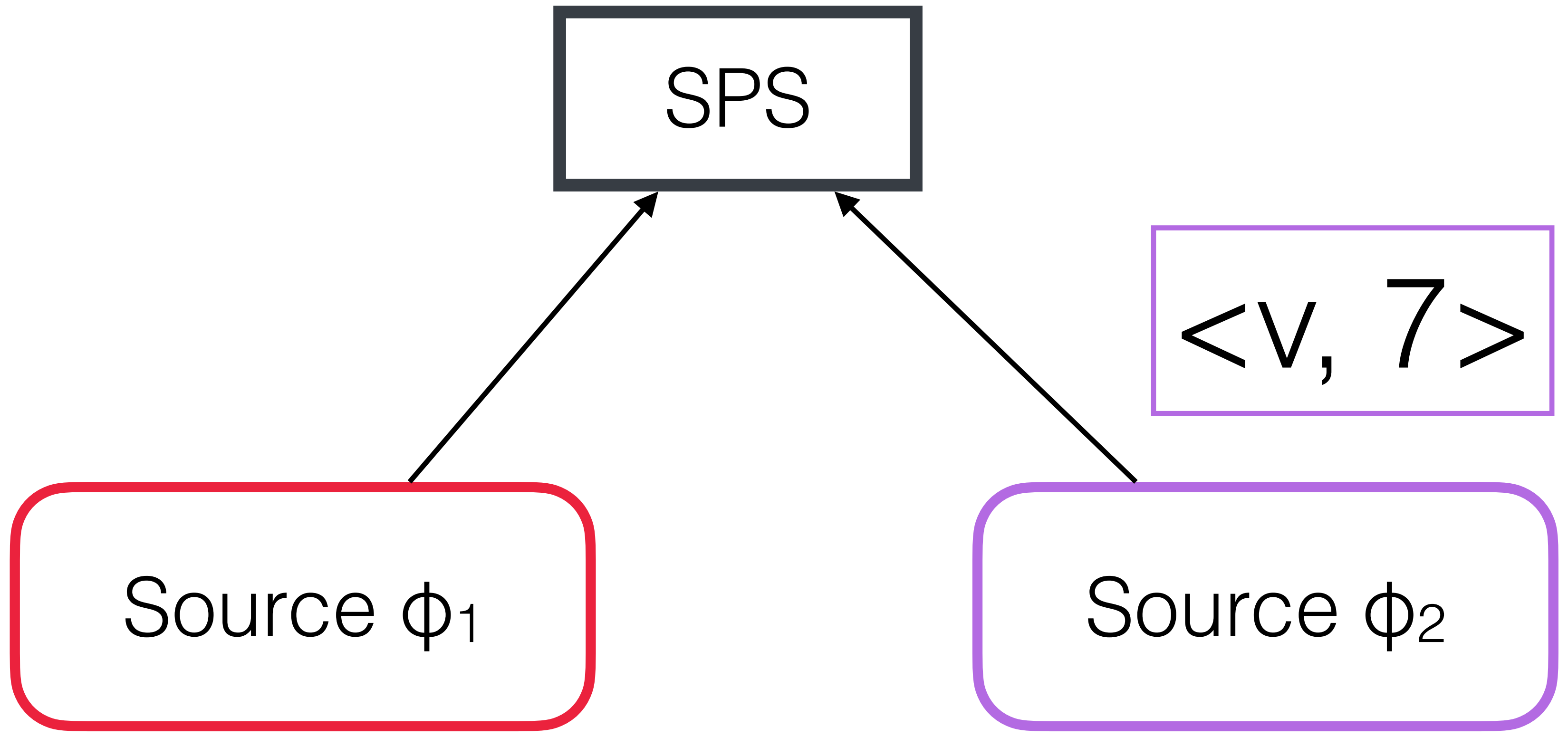
$$(t_{12}, \delta_{12}) = (2, 3)$$

Any tuple that ϕ_2 emits after c'=15 + 2 = 17
will have τ' > τ - δ = 8 - 3 = 5

Skew bound: example



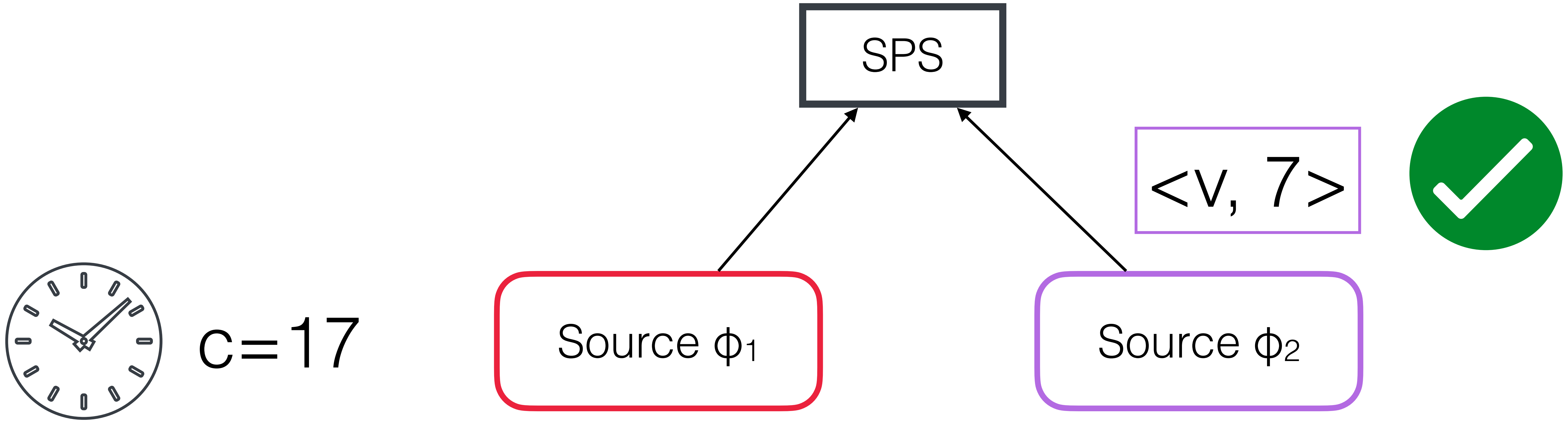
c=17



$$(t_{12}, \delta_{12}) \;=\; (2, 3)$$

Any tuple that ϕ_2 emits after c'=15 + 2 = 17
will have τ' > τ - δ = 8 - 3 = 5

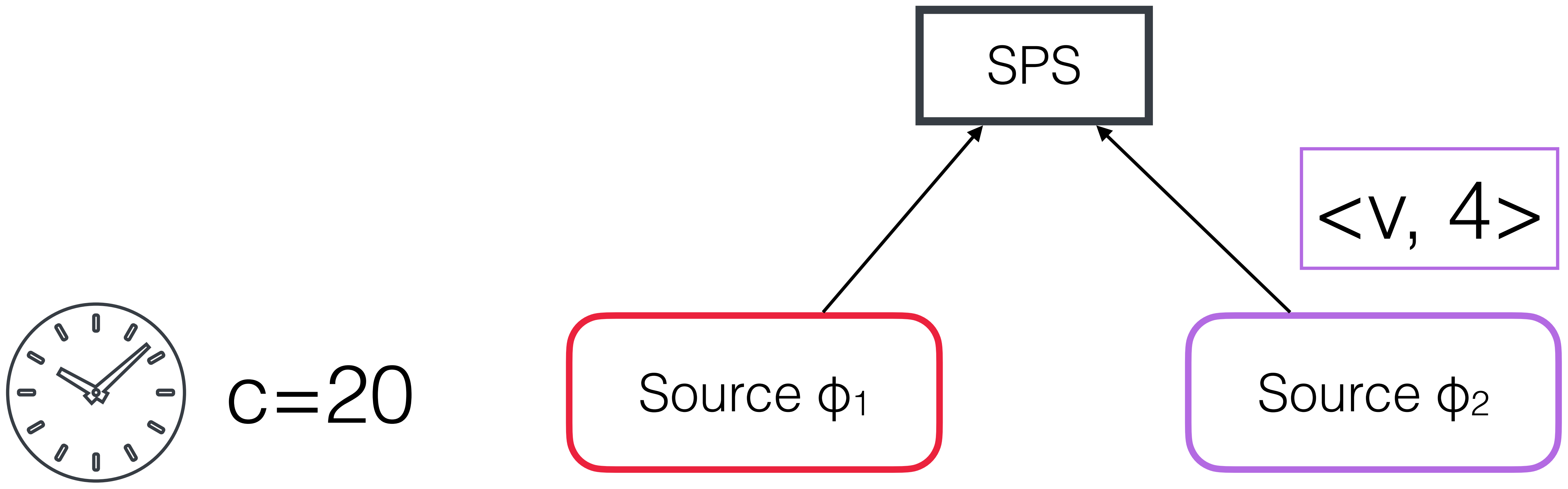
Skew bound: example



$$(t_{12}, \delta_{12}) \;=\; (2,\, 3)$$

Any tuple that ϕ_2 emits after c'=15 + 2 = 17
will have τ' > τ - δ = 8 - 3 = 5

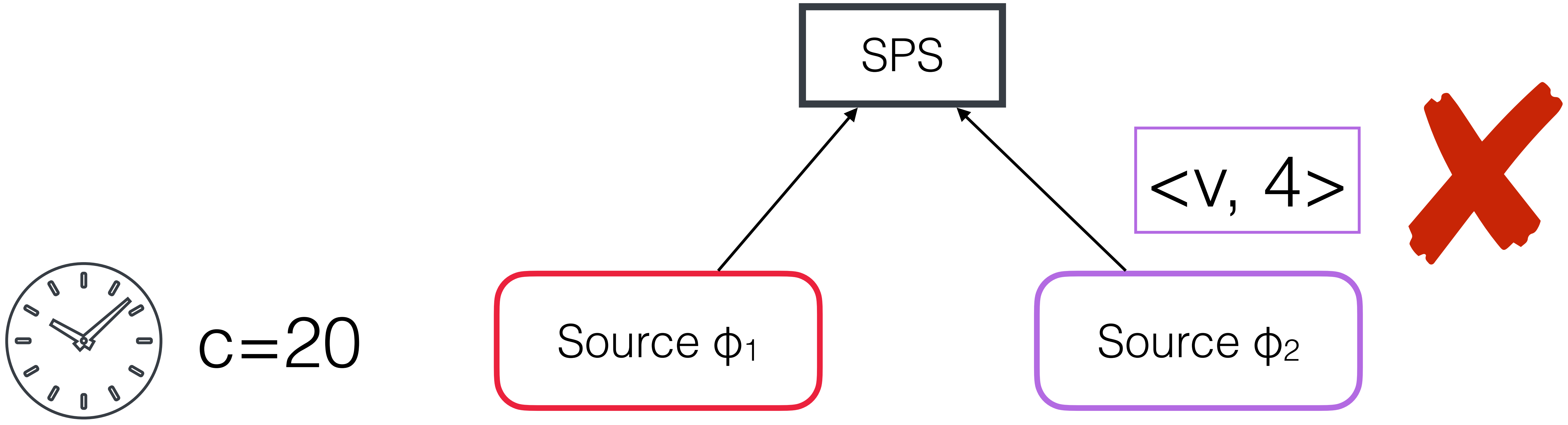
Skew bound: example



$$(t_{12}, \delta_{12}) = (2, 3)$$

Any tuple that ϕ_2 emits after c'=15 + 2 = 17
will have $\tau' > \tau - \delta = 8 - 3 = 5$

Skew bound: example



$$(t_{12}, \delta_{12}) = (2, 3)$$

Any tuple that ϕ_2 emits after $c'=15 + 2 = 17$
will have $\tau' > \tau - \delta = 8 - 3 = 5$

Out-of-order generation bound

How out-of-order a source ϕ_i generates tuples

is given by the skew bound of ϕ_i with respect to *itself*

i.e., (t_{ii}, δ_{ii})

The reordering off timestamps is *bounded* by δ_{ii}.

Out-of-order generation bound

How out-of-order a source ϕ_i generates tuples

is given by the skew bound of ϕ_i with respect to *itself*

i.e., (t_{ii}, δ_{ii})

The reordering off timestamps is *bounded* by δ_{ii}.

What is the value of δ_{ii}

if timestamps are in order?

if there are duplicate timestamps but no reordering?

Latency bound

The bound on transmission latency from ϕ_i to the stream processor is given by L_i units of wall-clock time.

If any tuple from ϕ_i takes t units of processing time to be transmitted to the stream processor, then

$$0 \leq t \leq L_i .$$

Skew bound matrix

$$
\begin{array}{cccc}
 & \phi_1 & \phi_2 & \phi_3 \\
B = & \left(\begin{array}{ccc} (0,0) & (1,1) & (1,3) \\ - & (0,1) & (1,1) \\ - & - & (0,2) \end{array}\right) & & \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array}
\end{array}
$$

Skew bound matrix

$$\begin{array}{cc} & \begin{array}{ccc} \phi_1 & \phi_2 & \phi_3 \end{array} \\ B = & \begin{pmatrix} (0,0) & (1,1) & \boxed{(1,3)} \\ - & (0,1) & (1,1) \\ - & - & (0,2) \end{pmatrix} \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array} \end{array}$$

ϕ_3 lags behind ϕ_1
by at most 3 units of event time
and this guarantee is delayed by
1 unit of processing time.

Skew bound matrix

$$\begin{array}{cc} & \begin{array}{ccc} \phi_1 & \phi_2 & \phi_3 \end{array} \\ B = & \begin{pmatrix} (0,0) & (1,1) & \boxed{(1,3)} \\ - & \boxed{(0,1)} & (1,1) \\ - & - & (0,2) \end{pmatrix} \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array} \end{array}$$

ϕ_3 lags behind ϕ_1 by at most 3 units of event time and this guarantee is delayed by 1 unit of processing time.

ϕ_2 allows duplicate timestamps

Heartbeat generation algorithm

Input: Skew-bound matrix B, Latency bounds $L_1, L_2, \ldots, L_n$

Output: Heartbeats for S_i in array τ_i

1. $\tau_i[0] =$ minimum application time $- 1 \quad \forall i \in \{1, \ldots, n\}$
2. When a tuple with timestamp τ arrives on S_i at time c:
3. for $j = 1$ to n do
4. $\quad \tau_j[c + t_{ij} + L_j] = \max(\tau_j[c + t_{ij} + L_j], \tau - \delta_{ij})$

Indirect guarantees

$$
B = \begin{array}{cl}
\begin{array}{ccc} \quad\phi_1 \quad & \quad \phi_2 \quad & \quad \phi_3 \quad \end{array} & \\
\begin{pmatrix} (0,0) & (1,1) & (1,3) \\ - & (0,1) & (1,1) \\ - & - & (0,2) \end{pmatrix} & \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array}
\end{array}
$$

Indirect guarantees

$$\begin{array}{cccc} \phi_1 & \phi_2 & \phi_3 & \\ \end{array}$$

$$B = \begin{pmatrix} (0,0) & (1,1) & (1,3) \\ - & (0,1) & \boxed{(1,1)} \\ - & - & (0,2) \end{pmatrix} \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array}$$

ϕ_3 lags behind ϕ_2
by 1 unit of event time

Indirect guarantees

$$B = \begin{array}{cc} & \begin{array}{ccc} \phi_1 & \phi_2 & \phi_3 \end{array} \\ & \left(\begin{array}{ccc} (0,0) & \boxed{(1,1)} & (1,3) \\ - & (0,1) & \boxed{(1,1)} \\ - & - & (0,2) \end{array}\right) \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array} \end{array}$$

ϕ_3 lags behind ϕ_2 by 1 unit of event time

ϕ_2 lags behind ϕ_1 by 1 unit of event time

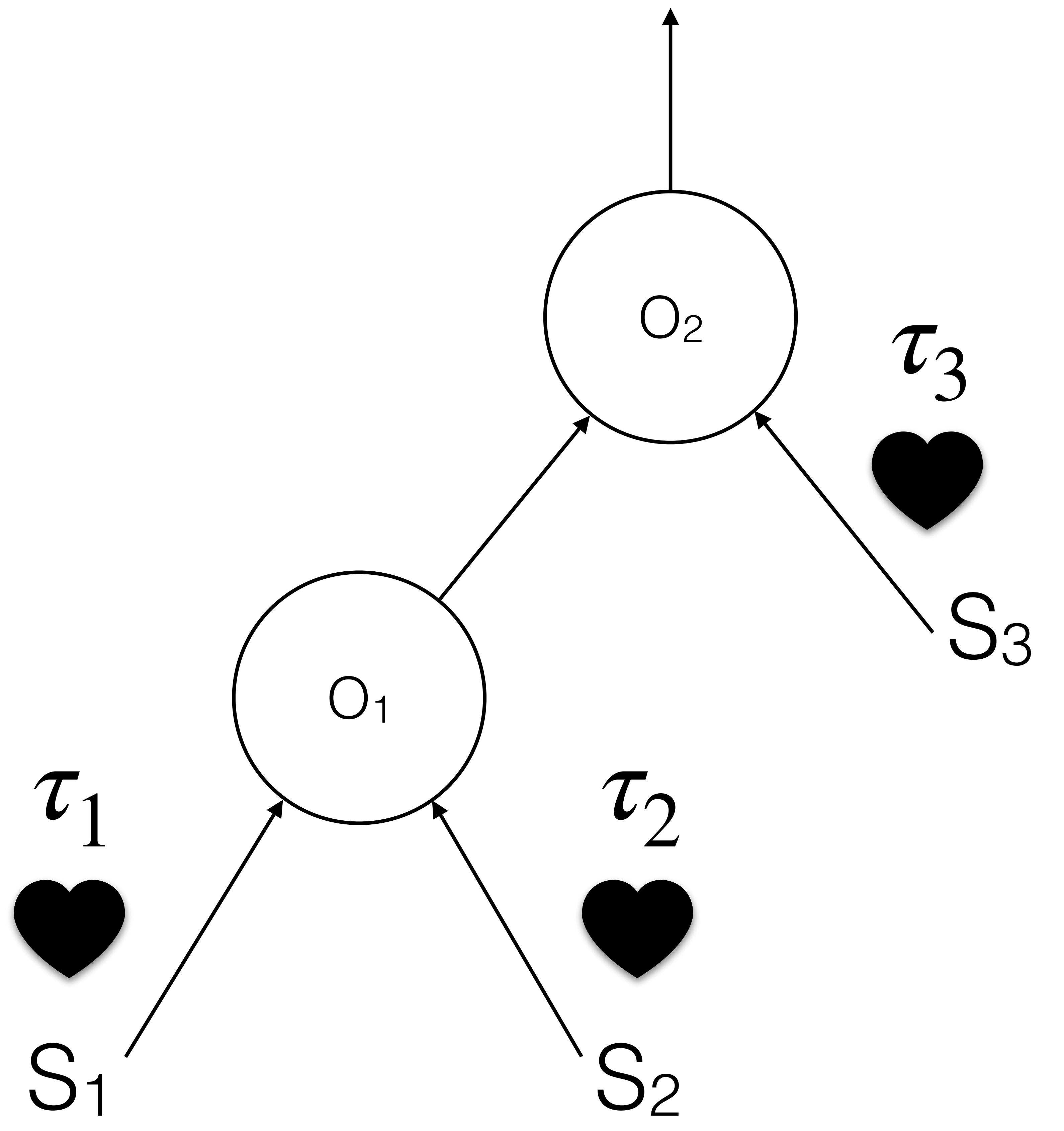
Indirect guarantees

$$\begin{array}{cccc} & \phi_1 & \phi_2 & \phi_3 & \\ B = & \left(\begin{array}{ccc} (0,0) & \boxed{(1,1)} & \boxed{(1,2)} \\ - & (0,1) & \boxed{(1,1)} \\ - & - & (0,2) \end{array}\right) & \begin{array}{c} \phi_1 \\ \phi_2 \\ \phi_3 \end{array} \end{array}$$

ϕ_3 lags behind ϕ_2 by 1 unit of event time

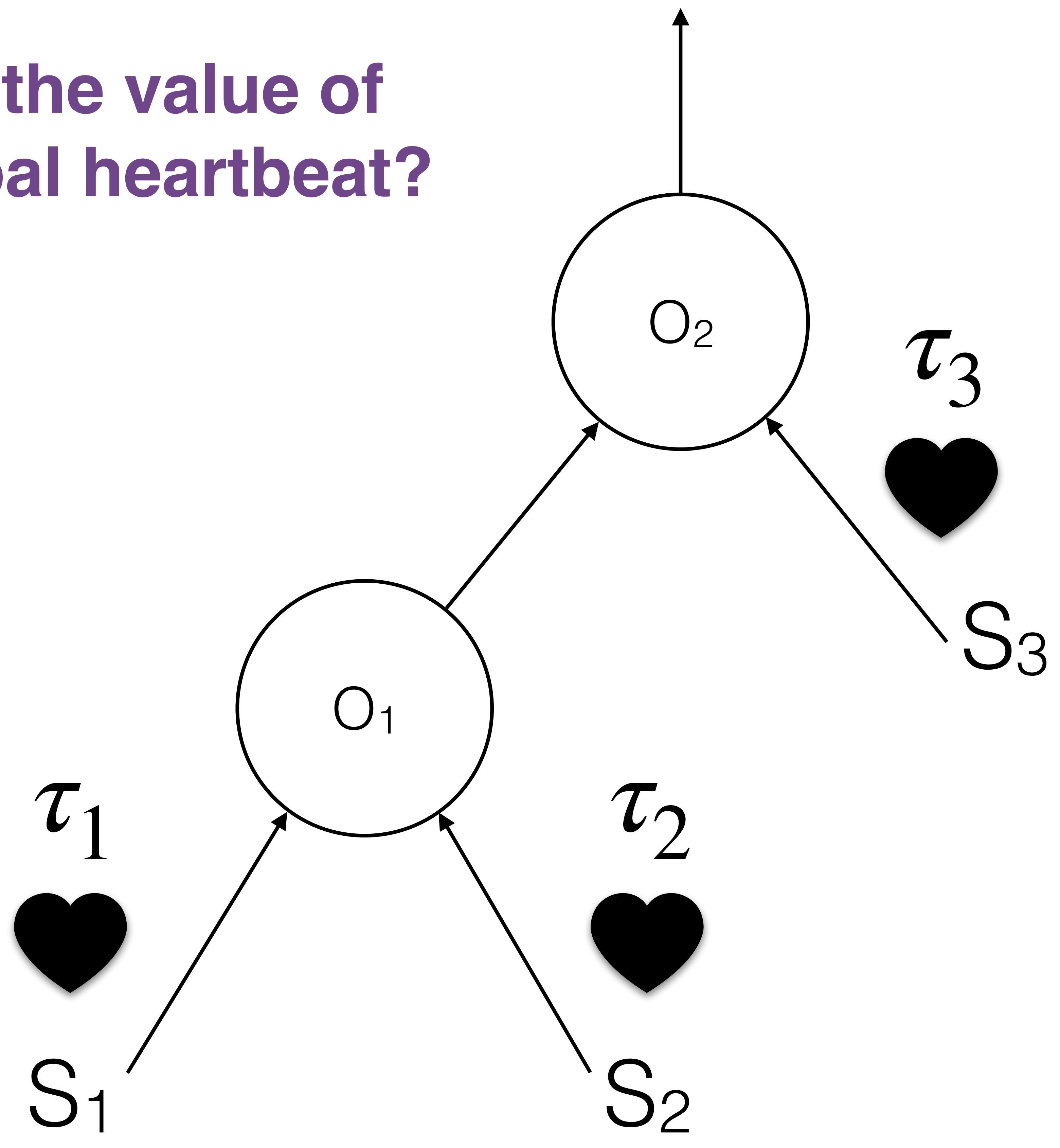
ϕ_2 lags behind ϕ_1 by 1 unit of event time

Query-level heartbeats



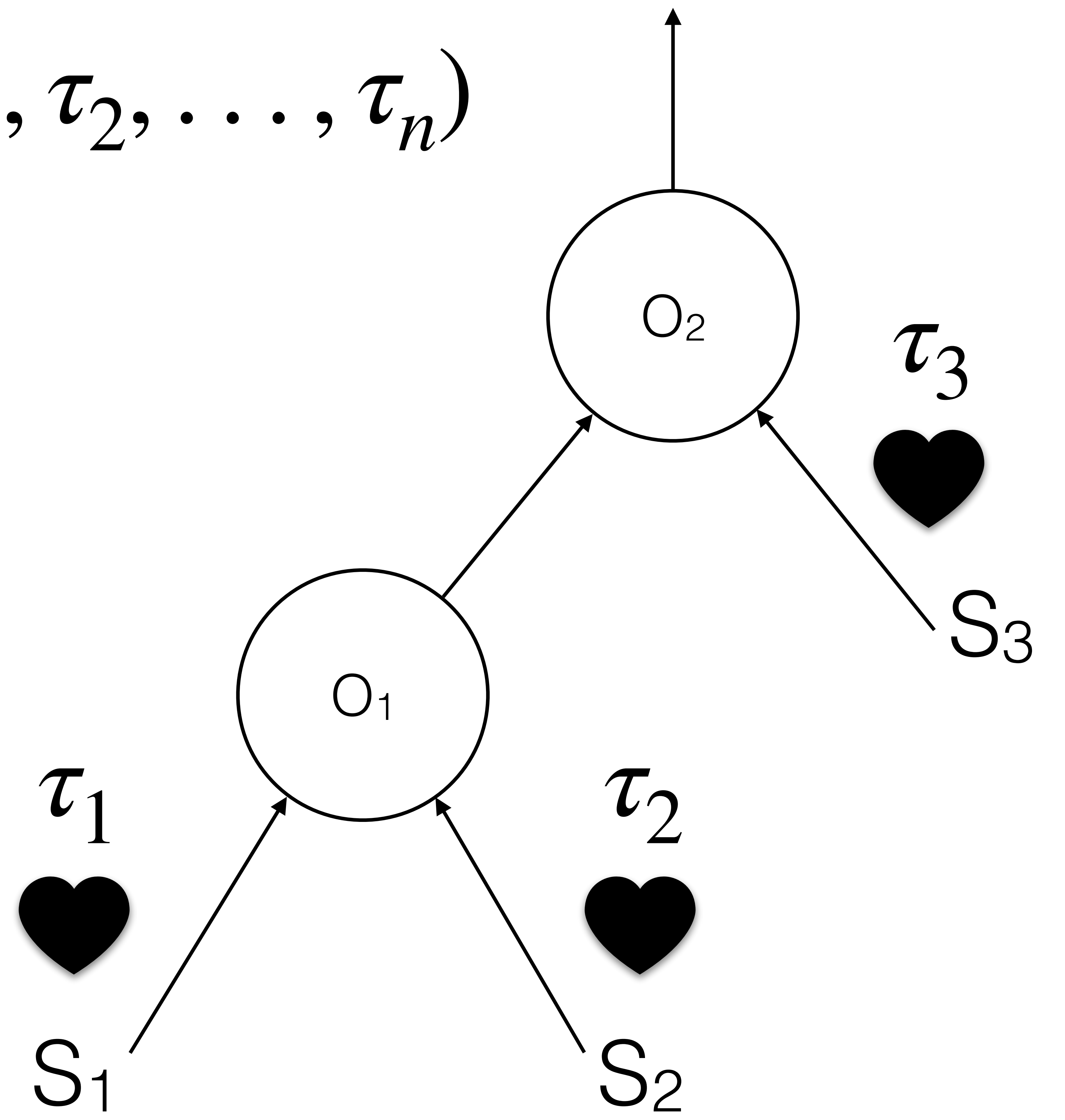
Query-level heartbeats

What is the value of the global heartbeat?



Query-level heartbeats

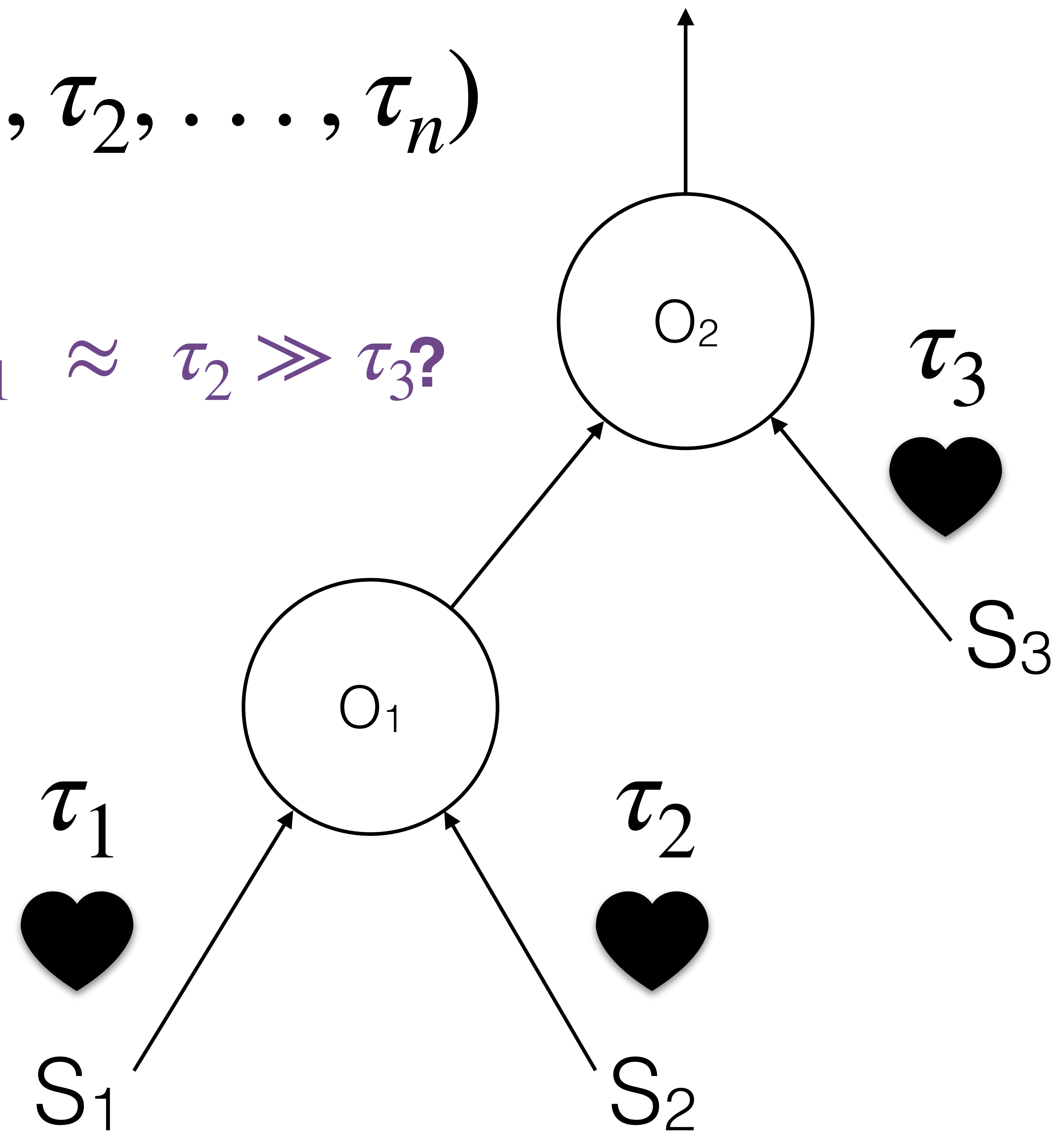
$$\tau = min(\tau_1, \tau_2, \ldots, \tau_n)$$



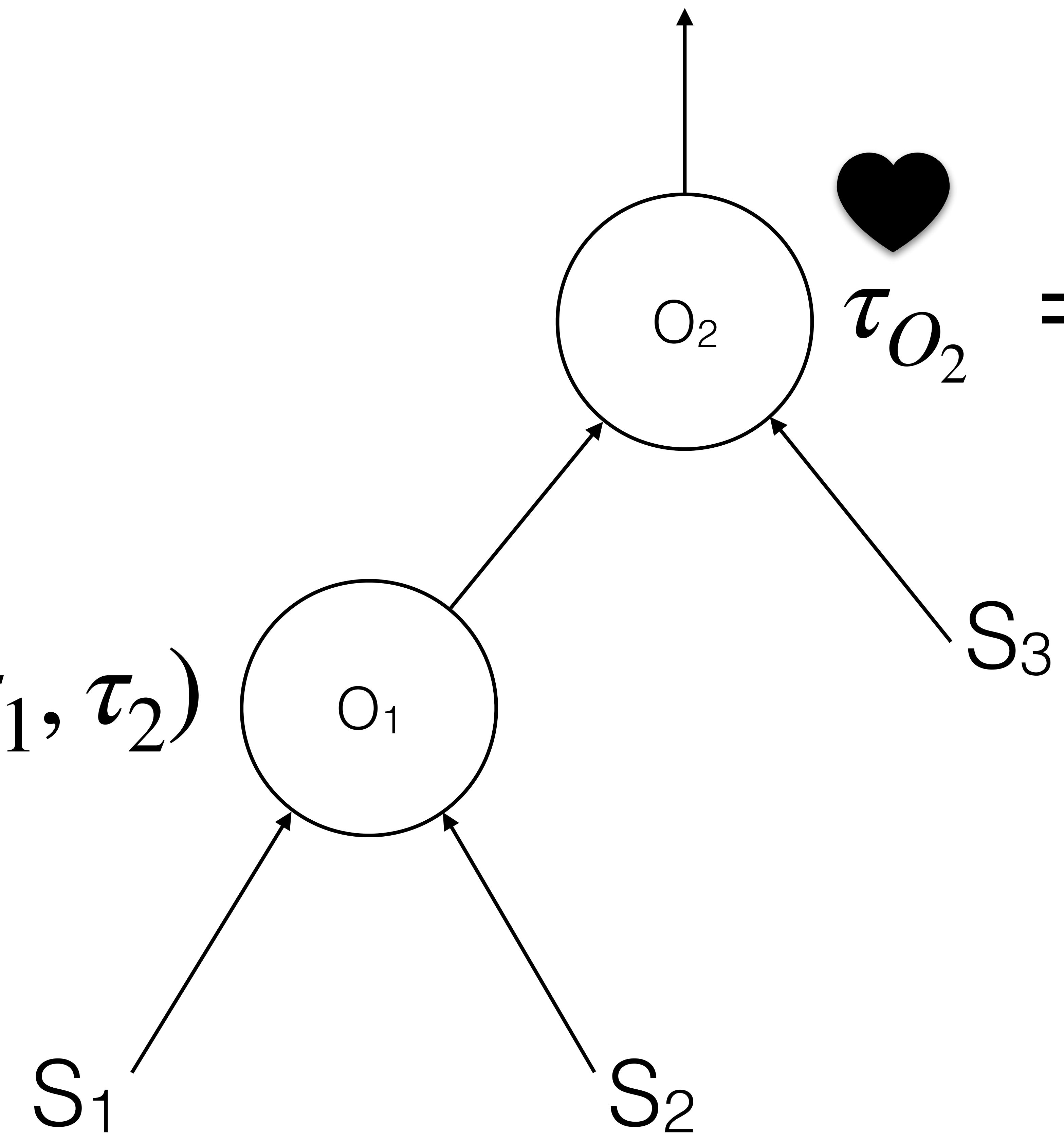
Query-level heartbeats

$$\tau \ = min(\tau_1, \tau_2, \ldots, \tau_n)$$

What if $\tau_1 \ \approx \ \tau_2 \gg \tau_3$**?**



Operator-level heartbeats



$$\tau_{O_2} = min(\tau_{O_1}, \tau_3)$$

$$\tau_{O_1} = min(\tau_1, \tau_2)$$

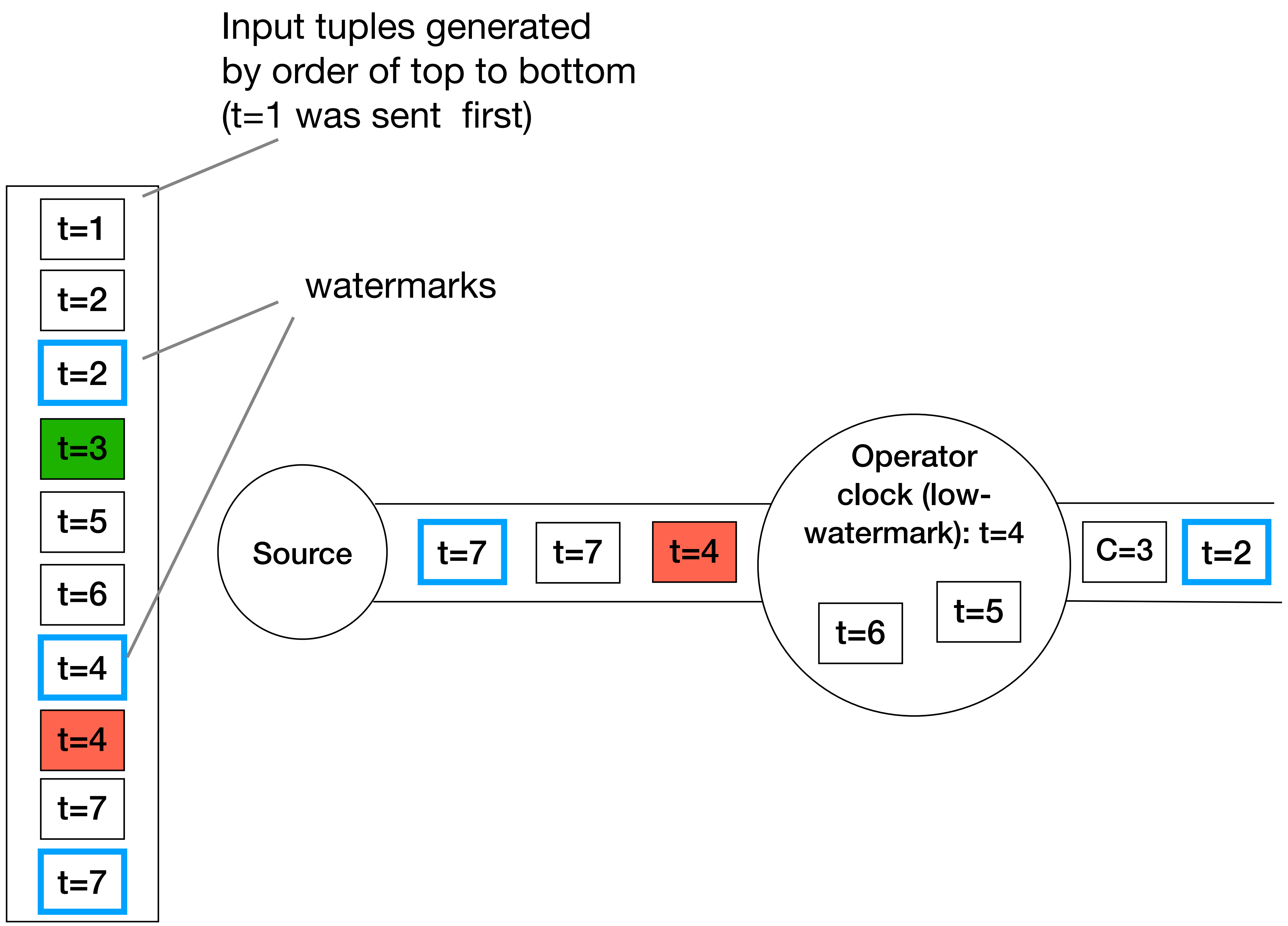
Watermarks

Low watermark

- The low watermark for an attribute of a stream is the lowest value of that attribute within a certain subset of the stream.
- Future tuples will probabilistically bear a higher value than the current low-watermark for the same attribute.
- The mechanism is used by a streaming system to process data past the low watermark for an attribute, e.g. an aggregate grouped by the attribute, or to remove state that is maintained for the attribute, for instance, the corresponding hash table entries of a hash join computation.

Low watermark in action

- Close first window [0,4) when low-watermark t=4 arrives
- Normally the window would close when t=5 arrives, but because the low watermark reflects the oldest pending work in the system, it is the low-watermark that closes windows to cater for late data.
- The window will output C=3 for t=1, t=2, and t=3
- Drop t=4 because it is not greater (more recent) than the low-watermark



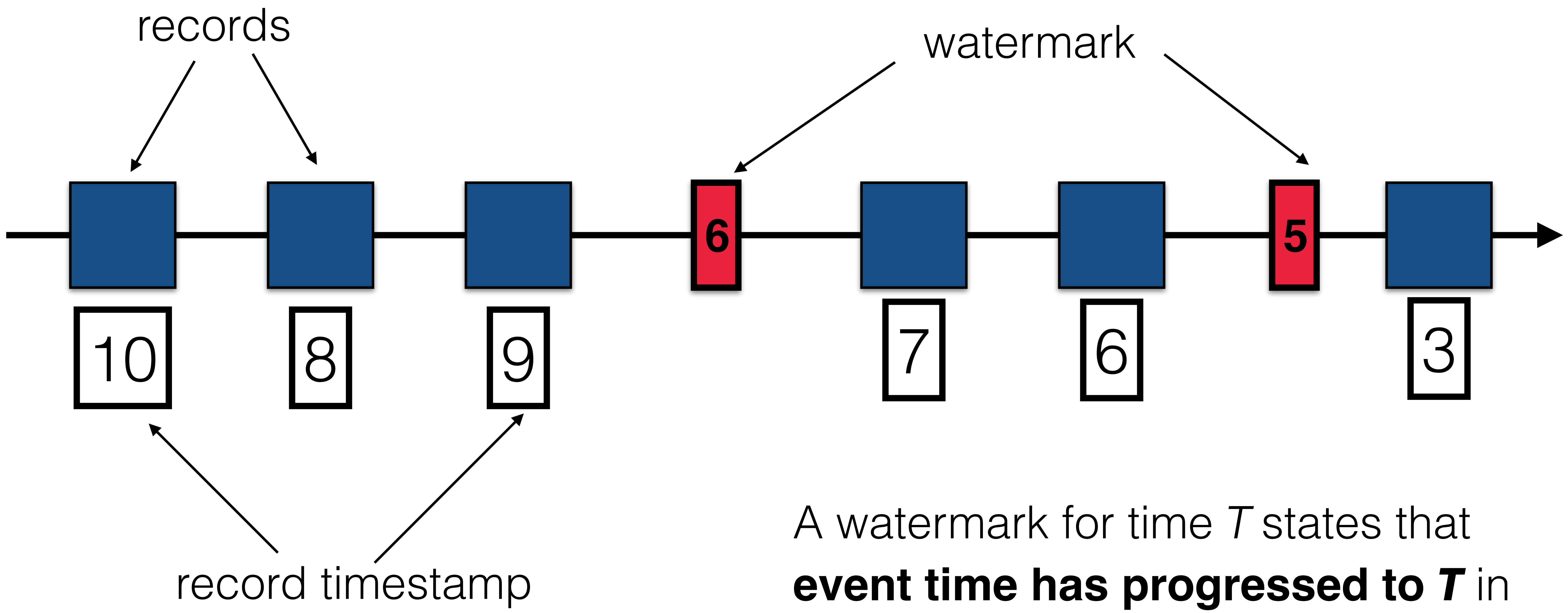
Slack vs heartbeats

- Heartbeats and slack are both *external* to a data stream.
- Heartbeats are signals communicated from an input source to a streaming system's ingestion point.
- Differently to heartbeats, which is a mechanism of the streaming system hidden from users, slack is part of the query specification provided by users.

Heartbeats vs low watermark

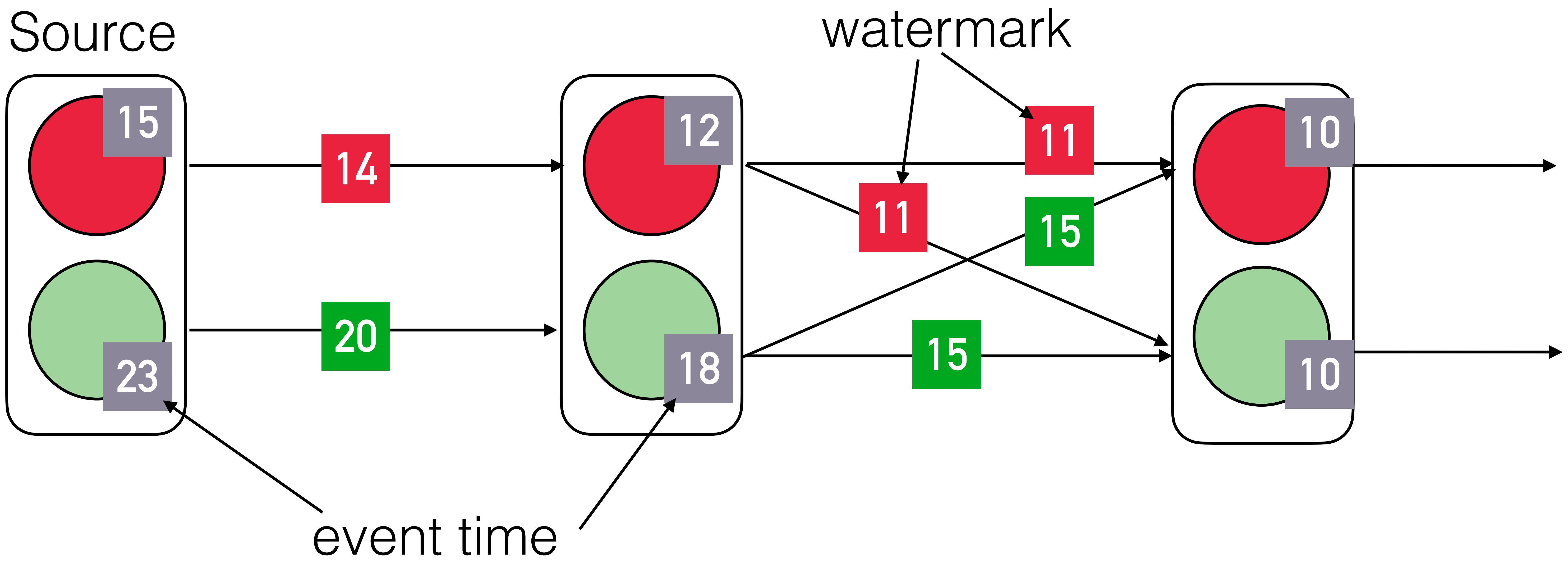
- Heartbeats and low-watermarks are similar in terms of progress-tracking logic.
- While heartbeats address the progress of stream tuple generation at the input sources, the low-watermark extends this to the processing progress of computations in the streaming system by reflecting their oldest pending work.
- The low-watermark generalizes the concept of the oldest value, which signifies the current progress point, to any progressing attribute of a stream tuple besides timestamps.

Watermarks (in Flink) flow along dataflow edges.
They are **special records** generated by the sources or assigned by the application.



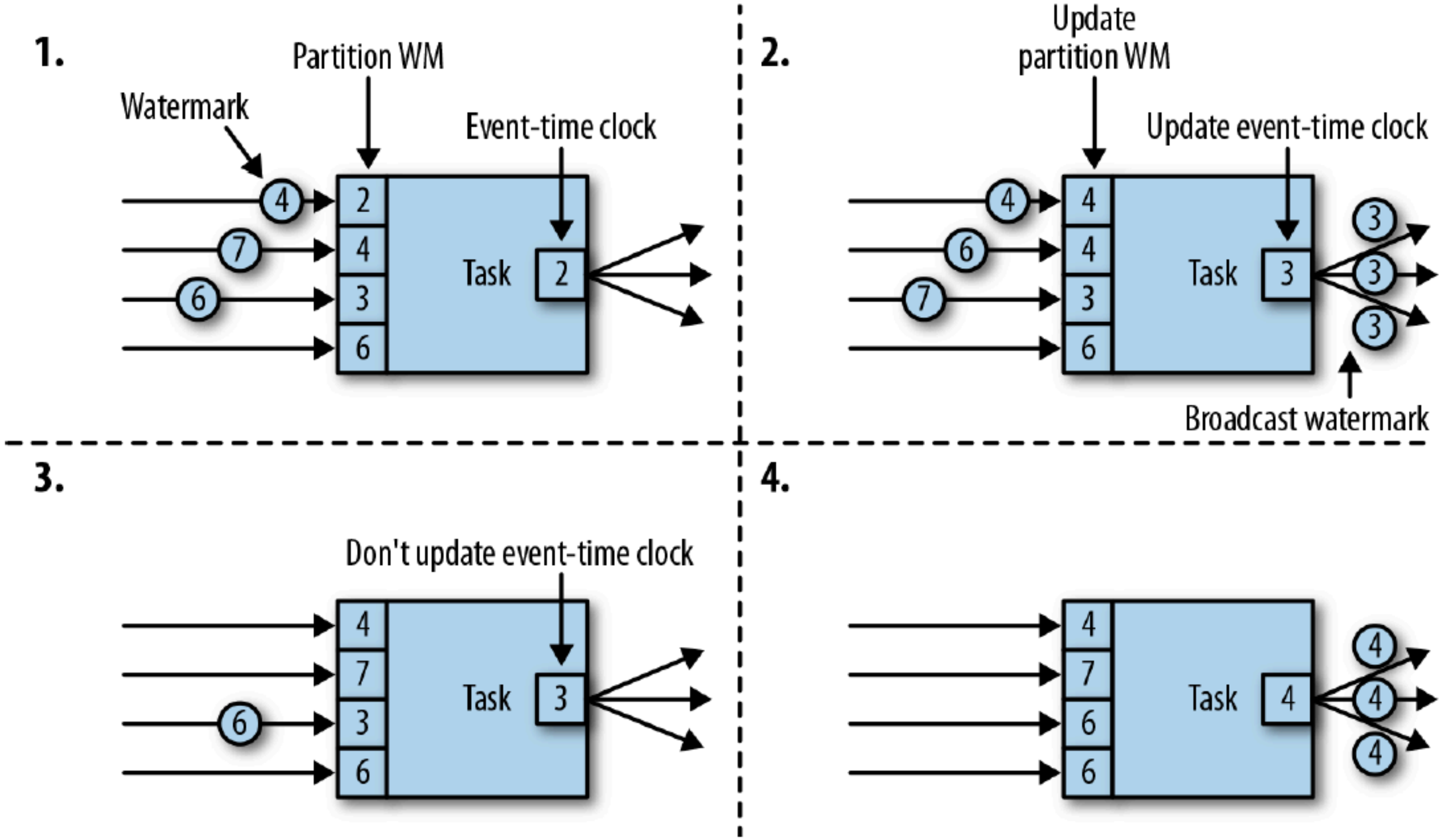
A watermark for time *T* states that **event time has progressed to *T*** in that particular stream (or partition).

Watermark propagation



- The *input* watermark captures the progress of upstream stages
  - minimum of output watermarks of all upstream tasks
- The *output* watermark captures the progress of the stage itself
  - minimum of input watermarks and event-times of non-late data

Event-time update



Watermark properties

1. Watermarks must be **monotonically increasing** in order to ensure that the event time clocks of tasks are progressing and not going backwards.

2. A watermark with a timestamp *T* indicates that all subsequent records should have timestamps > *T*.

Evaluation of event-time windows

Watermarks are essential to both event-time windows and operators handling out-of-order events:

- When an operator receives a watermark with time *T*, it can assume that no further events with timestamp less than *T* will be received.
- It can then either trigger computation or order received events.

Trade-offs

Watermarks provide a configurable trade-off between **results confidence** and **latency**:

- *Eager* watermarks ensure low latency but provide lower confidence
  - Late events might arrive after the watermark
- *Slow* watermarks increase confidence but they might lead to higher processing latency.

Watermarks in Flink

Periodic: periodically ask the user-defined function for the current watermark timestamp.

Punctuated: check for a watermark in each passing record, e.g. if the stream contains special records that encode watermark information.

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// generate watermarks every 5 seconds
env.getConfig.setAutoWatermarkInterval(5000)
```

```scala
/**
 * This generator generates watermarks assuming that elements arrive out of order,
 * but only to a certain degree. The latest elements for a certain timestamp t will arrive
 * at most n milliseconds after the earliest elements for timestamp t.
 */
class BoundedOutOfOrdernessGenerator extends AssignerWithPeriodicWatermarks[MyEvent] {

    val maxOutOfOrderness = 3500L // 3.5 seconds

    var currentMaxTimestamp: Long = _

    override def onEvent(element: MyEvent, eventTimestamp: Long): Unit = {
        currentMaxTimestamp = max(eventTimestamp, currentMaxTimestamp)
    }

    override def onPeriodicEmit(): Unit = {
        // emit the watermark as current highest timestamp minus the out-of-orderness bound
        output.emitWatermark(new Watermark(currentMaxTimestamp - maxOutOfOrderness - 1));
    }
}
```

More examples: https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/event_timestamps_watermarks.html

Handling late data

- In many real-world applications, the system does not have enough knowledge to perfectly determine watermarks:
  - how long will a user might remain disconnected?
  - are they going through a tunnel, boarding a plane, or never playing again?
- Tracking global progress in a distributed system is problematic in the presence of *straggler* tasks.

What to do with late data?

- It is crucial that the stream processing system provides some mechanism to deal with events that might arrive *after the watermark*.
- Depending on the application requirements, you might want to:
  - ignore late data
  - log late data to some monitoring application
  - correct previously emitted results

```scala
val readings: DataStream[SensorReading] = ???

val countPer10Secs: DataStream[(String, Long, Int)] = readings
 .keyBy(_.id)
 .timeWindow(Time.seconds(10))
 // emit late readings to a side output
 .sideOutputLateData(new OutputTag[SensorReading]("late-readings"))
 // count readings per window
 .process(new CountFunction())
 // retrieve the late events from the side output as a stream

val lateStream: DataStream[SensorReading] = countPer10Secs
 .getSideOutput(new OutputTag[SensorReading]("late-readings"))
```

References

- D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDBJ*, 2003.
- P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 2003.
- U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*. ACM, 2004.
- J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high- performance stream systems. *In VLDB*, 2008.
- T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
- D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.
- T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, un- bounded, out-of-order data processing. *In VLDB*, 2015.
- P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.