Tutorial:

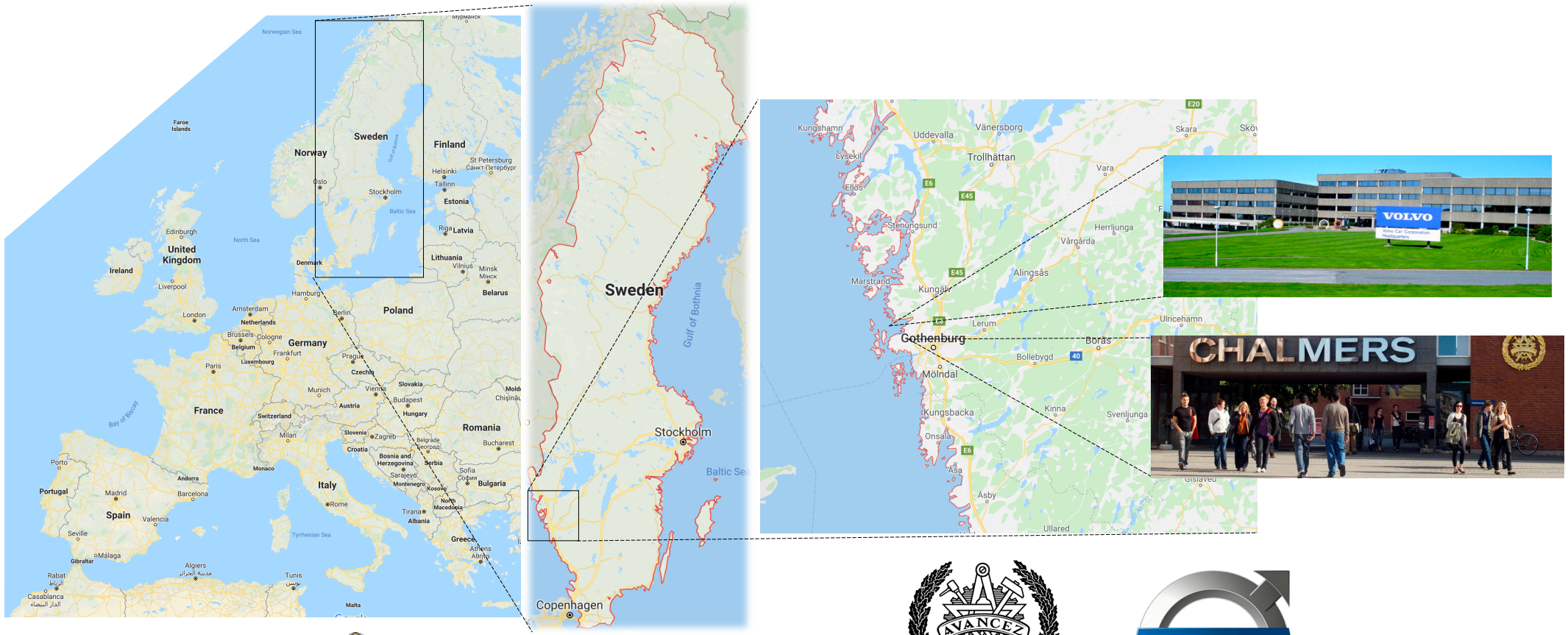
The Role of Event-Time Order in Data Streaming Analysis

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Bastian Havers
Chalmers University of Technology & Volvo Cars
Gothenburg, Sweden
havers@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se



Distributed Computing and Systems
Chalmers university of technology

CHALMERS
UNIVERSITY OF TECHNOLOGY

Agenda

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

https://github.com/ vincenzo- gulisano/debs2020_tutorial_event_time

Agenda

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

The era of **Big Data**

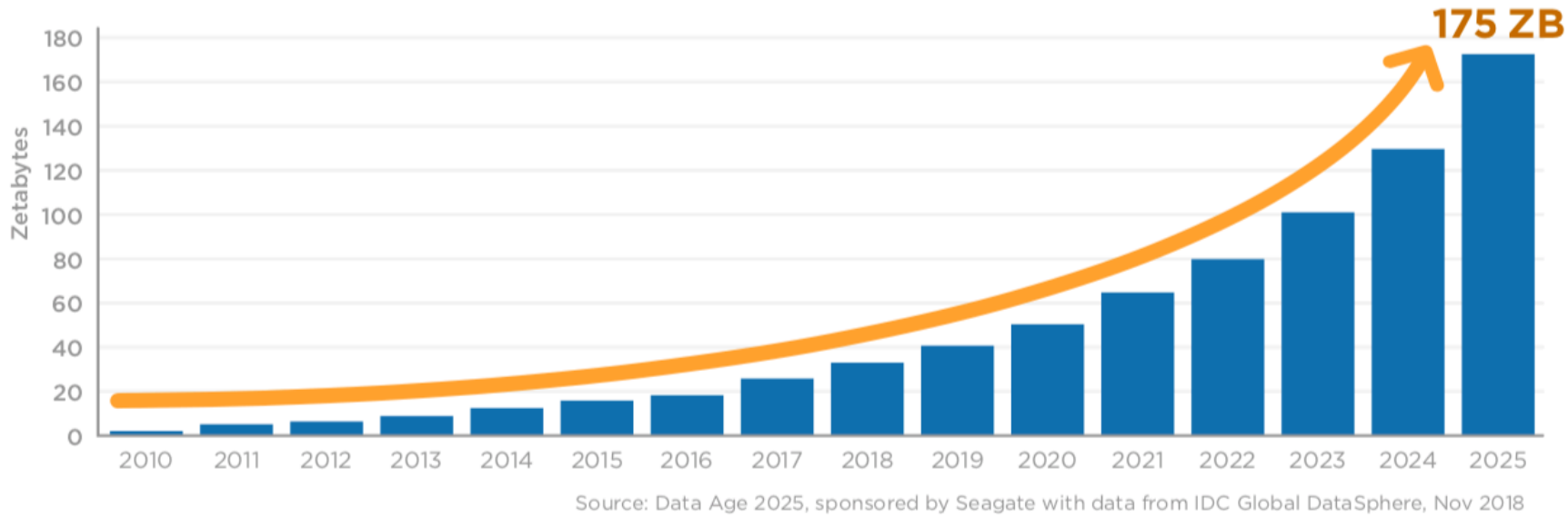
Terabyte Scale
10^{12} bytes

→

Zetabyte Scale
10^{9} Terabytes

1984

→



Source: Data Age 2025, sponsored by Seagate with data from IDC Global DataSphere, Nov 2018

Where does Big Data Originate?

219 billion photos
uploaded to Facebook

500 hours of video
uploaded to YouTube
every minute

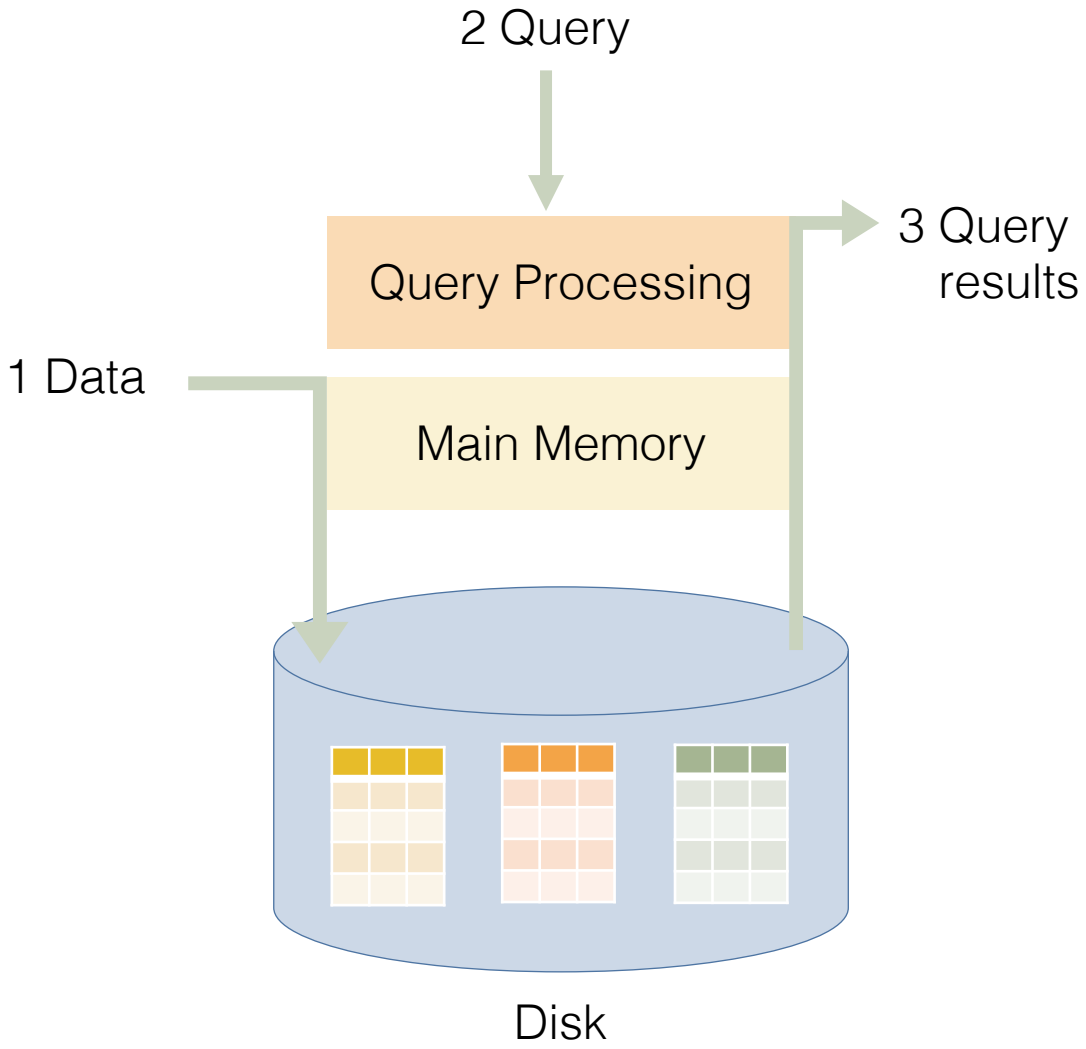
2.32 billion
Facebook users

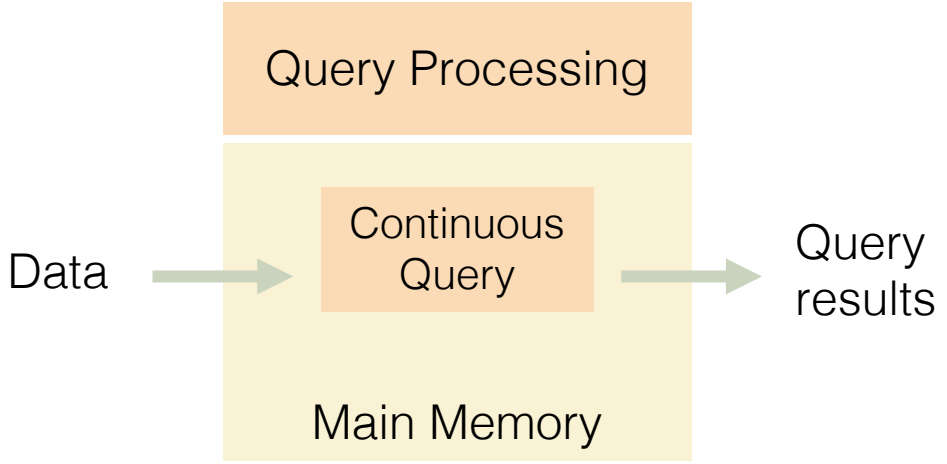
1 online interaction
every 18 seconds
by 2025

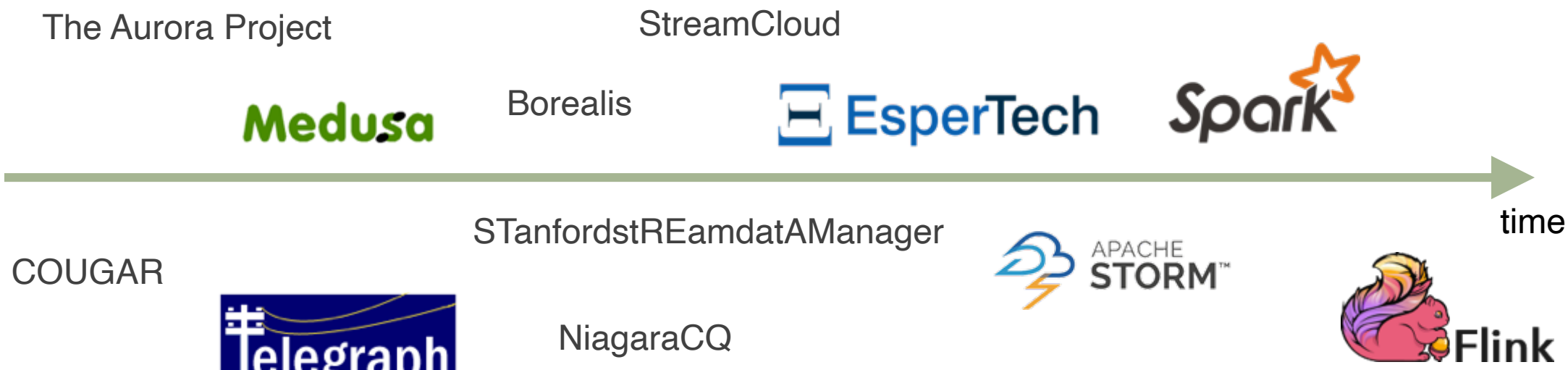
1 trillion sensors

in the **Internet of Things (IoT)**
by 2030

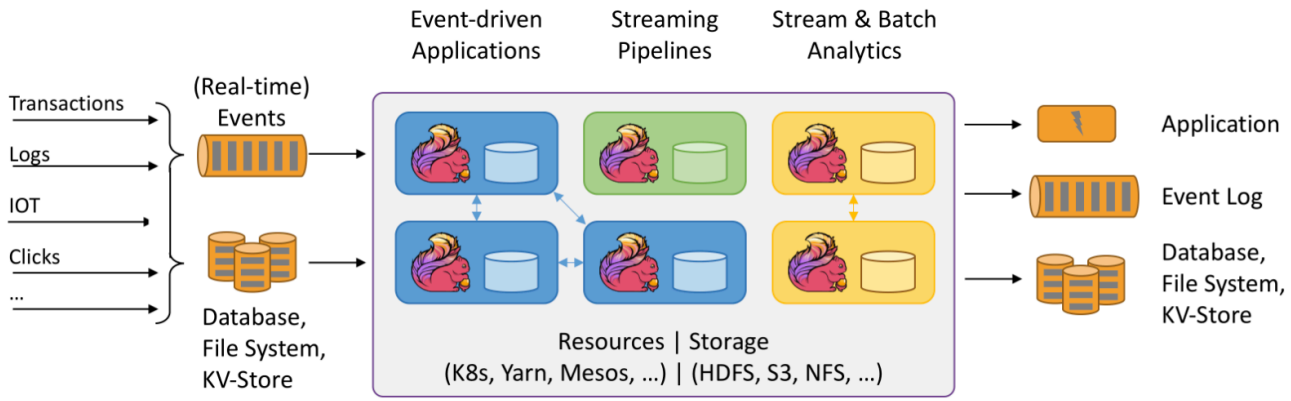
Database Management Systems (DBMSs) vs. Stream Processing Engines (SPEs)






The Aurora Project
StreamCloud
Medusa
Borealis
EsperTech
Spark
time
STanfordstREamdatAManager
COUGAR
APACHE
STORM
telegraph
Berkeley Database Research
NiagaraCQ
Flink

Apache Flink® — Stateful Computations over Data Streams



All streaming use cases

- Event-driven Applications
- Stream & Batch Analytics
- Data Pipelines & ETL

Learn more

Guaranteed correctness

- Exactly-once state consistency
- Event-time processing
- Sophisticated late data handling

Learn more

Layered APIs

- SQL on Stream & Batch Data
- DataStream API & DataSet API
- ProcessFunction (Time & State)

Learn more

Operational Focus

- Flexible deployment
- High-availability setup
- Savepoints

Learn more

Scales to any use case

- Scale-out architecture
- Support for very large state
- Incremental checkpointing

Learn more

Excellent Performance

- Low latency
- High throughput
- In-Memory computing

Learn more

Apache Flink

Software

Apache Flink is an open-source stream-processing framework developed by the Apache Software Foundation. The core of Apache Flink is a distributed streaming data-flow engine written in Java and Scala. Flink executes arbitrary dataflow programs in a data-parallel and pipelined manner. Wikipedia

Developed by: Apache Software Foundation

Stable release: 1.10.0 / February 11, 2020; 2 months ago

Initial release: May 2011; 9 years ago

License: Apache License 2.0

Written in: Java, Scala

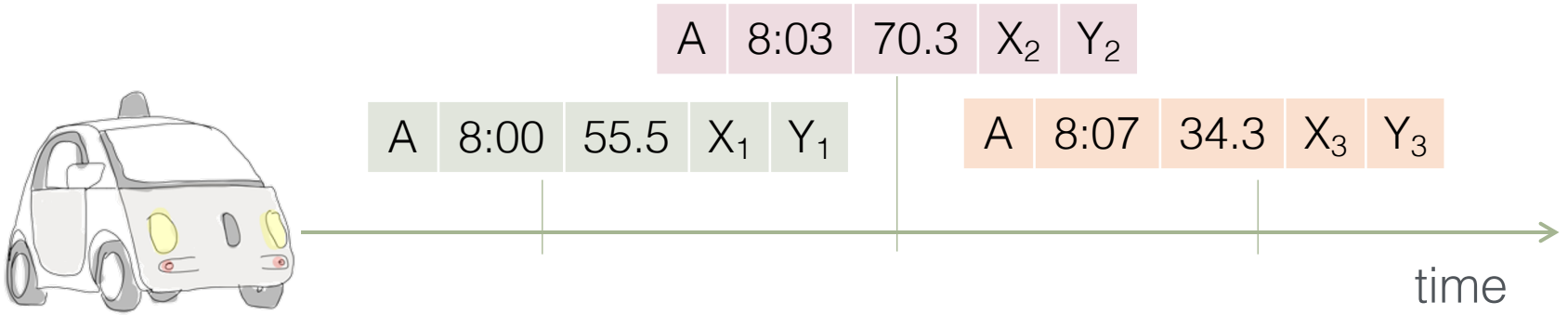
Flink-related images / code snippets in the following are taken from: https://flink.apache.org/

Data Stream:
unbounded sequence of tuples sharing the same schema

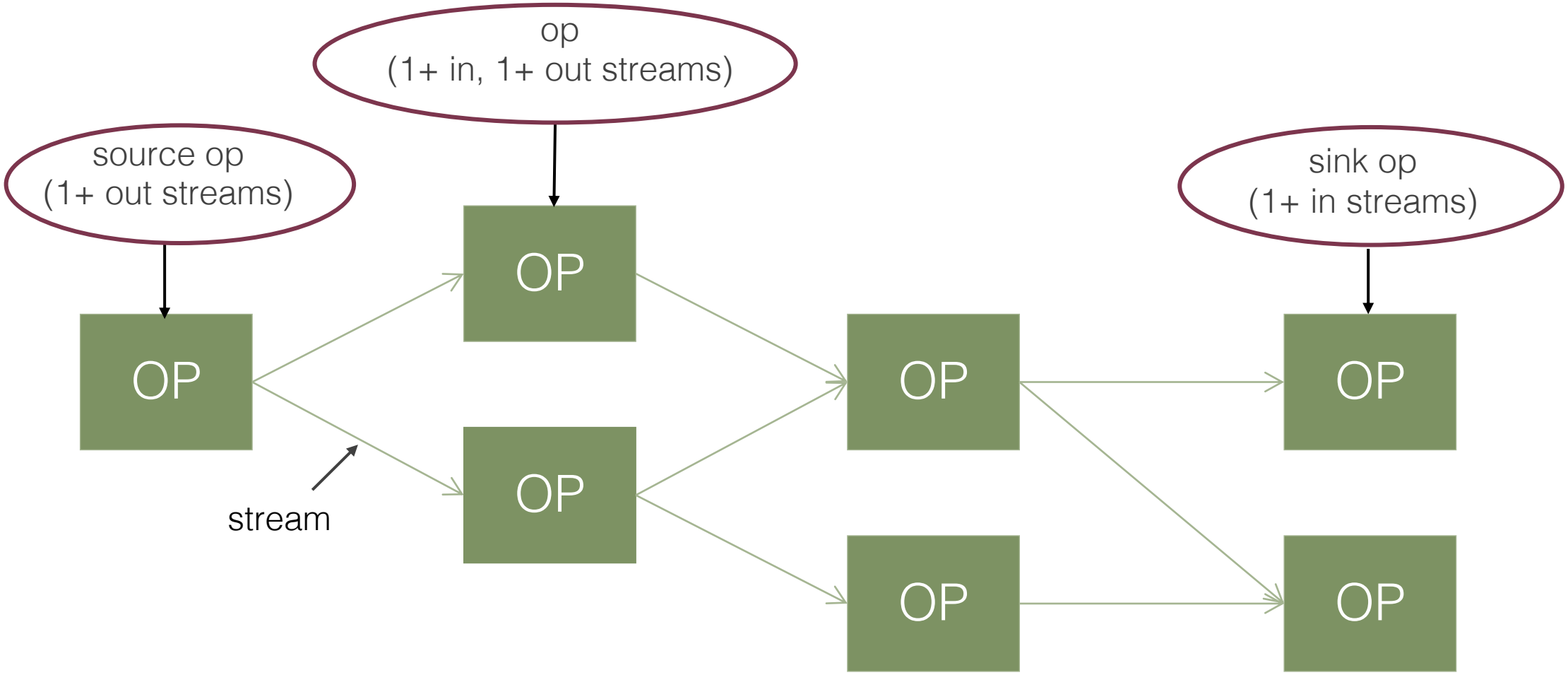
Example: vehicles' speed reports

Field	Field
vehicle id	text
time (secs)	text
speed (Km/h)	double
X coordinate	double
Y coordinate	double

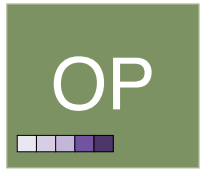
Let's assume each source (e.g., vehicle) produces and delivers a timestamp-sorted stream



Continuous query (or simply **query**): Directed Acyclic Graph (DAG) of streams and operators



Data Streaming **Operators**



Two main types:

- Stateless operators
  - do not maintain any state
  - one-by-one processing
  - if they maintain some state, such state does not evolve depending on the tuples being processed
- Stateful operators
  - maintain a state that evolves depending on the tuples being processed
  - produce output tuples that depend on multiple input tuples

Stateless Operators

Transform each tuple

Filter / route tuples based on one (or more) conditions

Map
DataStream → DataStream
	Takes one element and produces one element. A map function that doubles the values of the input stream:
	`DataStream<Integer> dataStream = //...`
	`dataStream.map(new MapFunction<Integer, Integer>() {`
	`  @Override`
	`  public Integer map(Integer value) throws Exception {`
	`  return 2 * value;`
	`  }`
	`});`

FlatMap
DataStream → DataStream
	Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:
	`dataStream.flatMap(new FlatMapFunction<String, String>() {`
	`  @Override`
	`  public void flatMap(String value, Collector<String> out)`
	`  throws Exception {`
	`  for(String word: value.split(" ")){`
	`  out.collect(word);`
	`  }`
	`  }`
	`});`

Filter
DataStream → DataStream
	Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:
	`dataStream.filter(new FilterFunction<Integer>() {`
	`  @Override`
	`  public boolean filter(Integer value) throws Exception {`
	`  return value != 0;`
	`  }`
	`});`

Stateful Operators

Aggregate information from multiple tuples (e.g., max, min, sum, ...)

Join tuples coming from 2 streams given a certain predicate

Aggregations
KeyedStream → DataStream
	Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).
	`keyedStream.sum(0);`
	`keyedStream.sum("key");`
	`keyedStream.min(0);`
	`keyedStream.min("key");`
	`keyedStream.max(0);`
	`keyedStream.max("key");`
	`keyedStream.minBy(0);`
	`keyedStream.minBy("key");`
	`keyedStream.maxBy(0);`
	`keyedStream.maxBy("key");`

Windows and **Stateful Analysis**

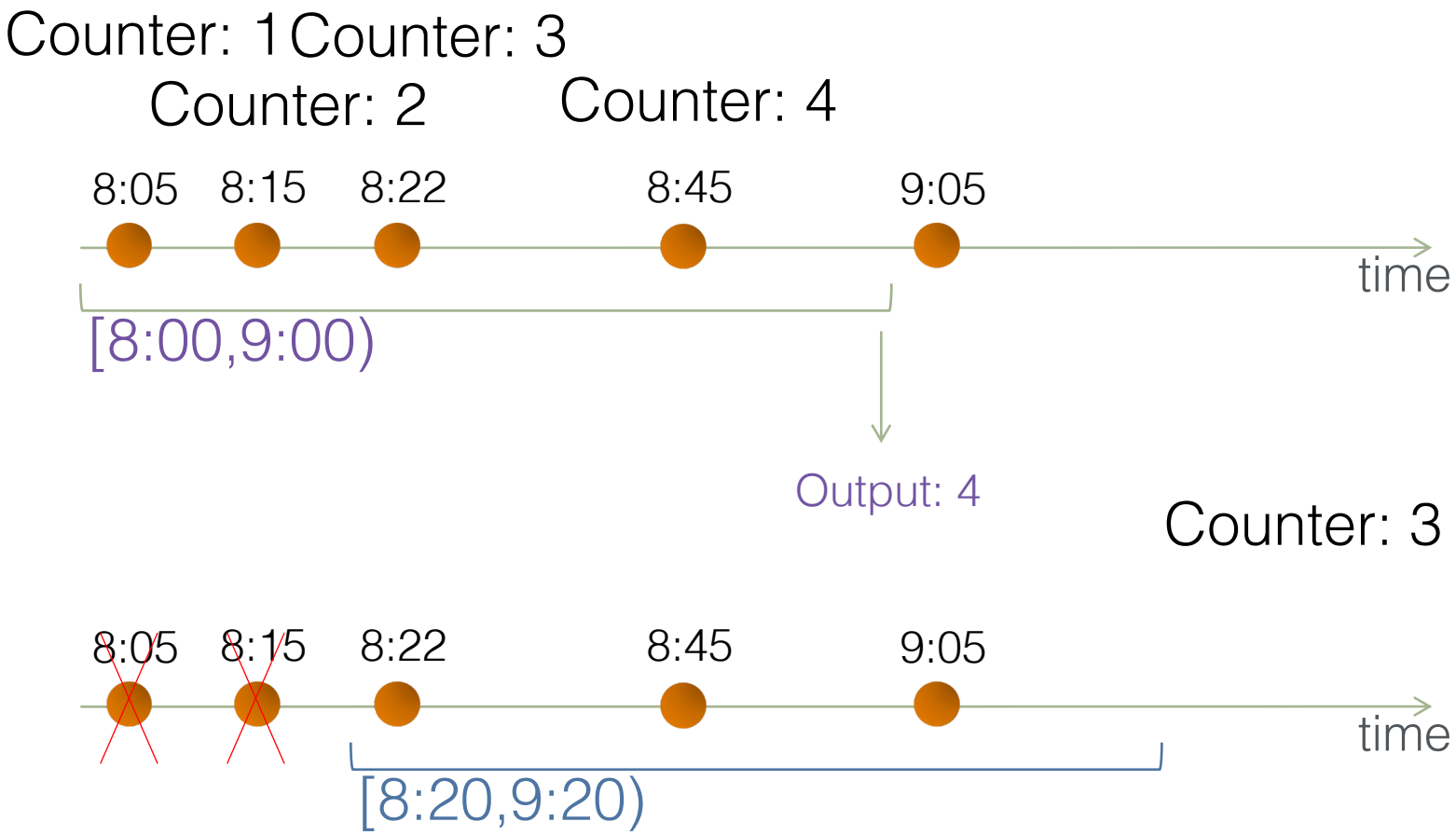
Stateful operations are done over windows:

- Time-based (e.g., tuples in the last 10 minutes)
- Tuple-based (e.g., given the last 50 tuples)

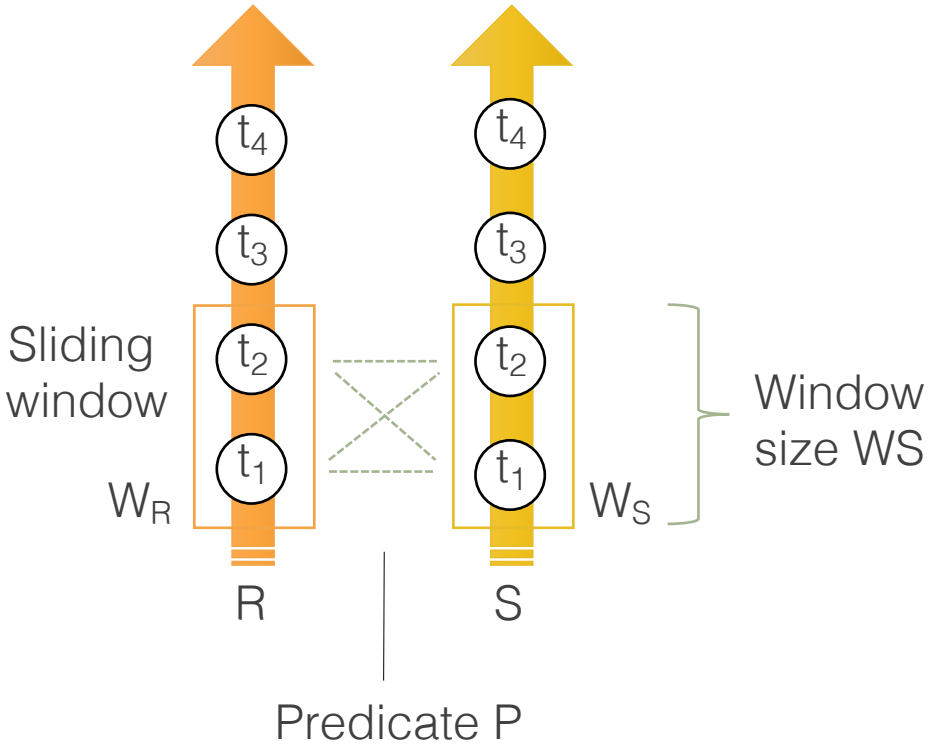
→ How many tuple in a window?
→ Which time period does a window span?

Example of time-based window of size 1 hour and advance 20 minutes

Time-based sliding window aggregation (count)



Time-based sliding window joining



Windows and **stateful analysis**

Window KeyedStream → WindowedStream	Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See windows for a complete description of windows. `dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5))); // Last 5 seconds of data`

Basic operators and **user-defined operators**

Besides a set of basic operators, SPEs usually allow the user to define ad-hoc operators (e.g., when existing aggregation are not enough)

Window Apply WindowedStream → DataStream AllWindowedStream → DataStream	Applies a general function to the window as a whole. Below is a function that manually sums the elements of a window. **Note:** If you are using a windowAll transformation, you need to use an AllWindowFunction instead.

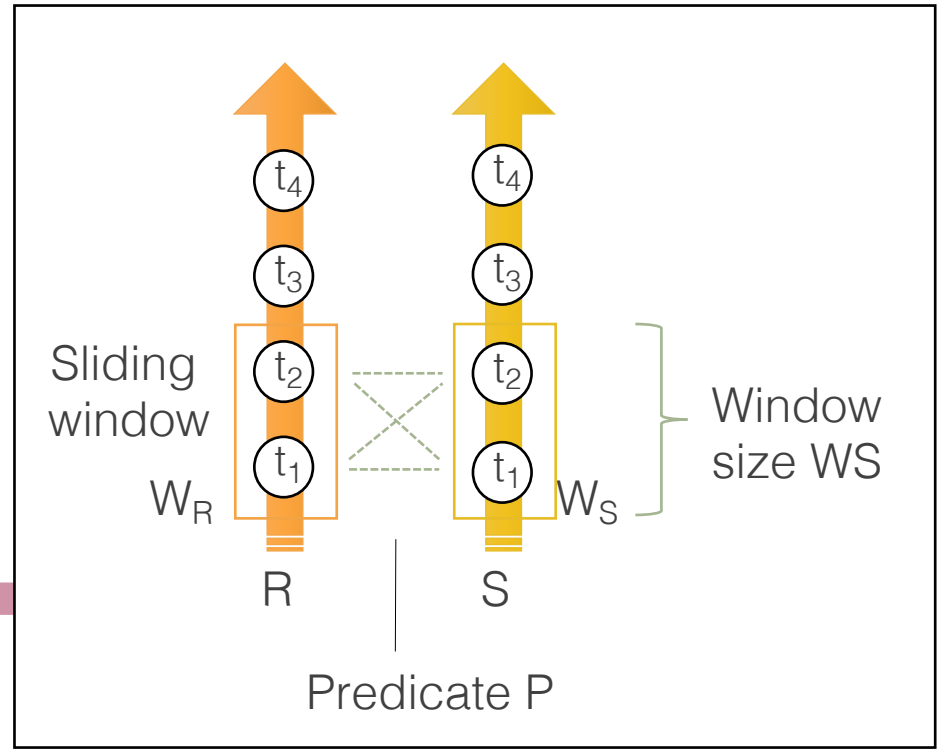
```
windowedStream.apply (new WindowFunction<Tuple2<String,Integer>, Integer, Tuple, W
indow>() {
    public void apply (Tuple tuple,
            Window window,
            Iterable<Tuple2<String, Integer>> values,
            Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});
```

SPEs and operators' variants

- Each SPE might define its own variants of certain streaming operators:

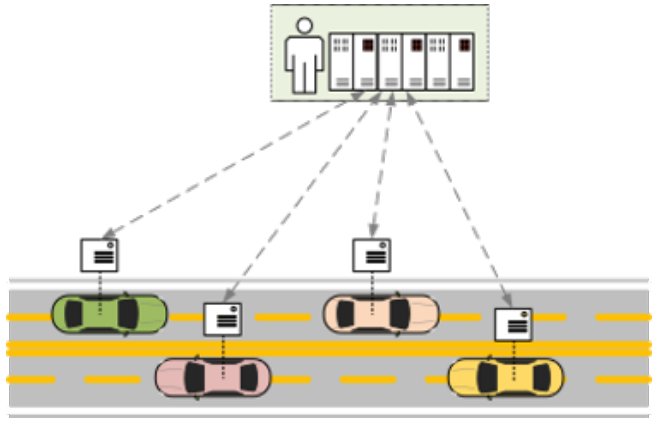
Joining

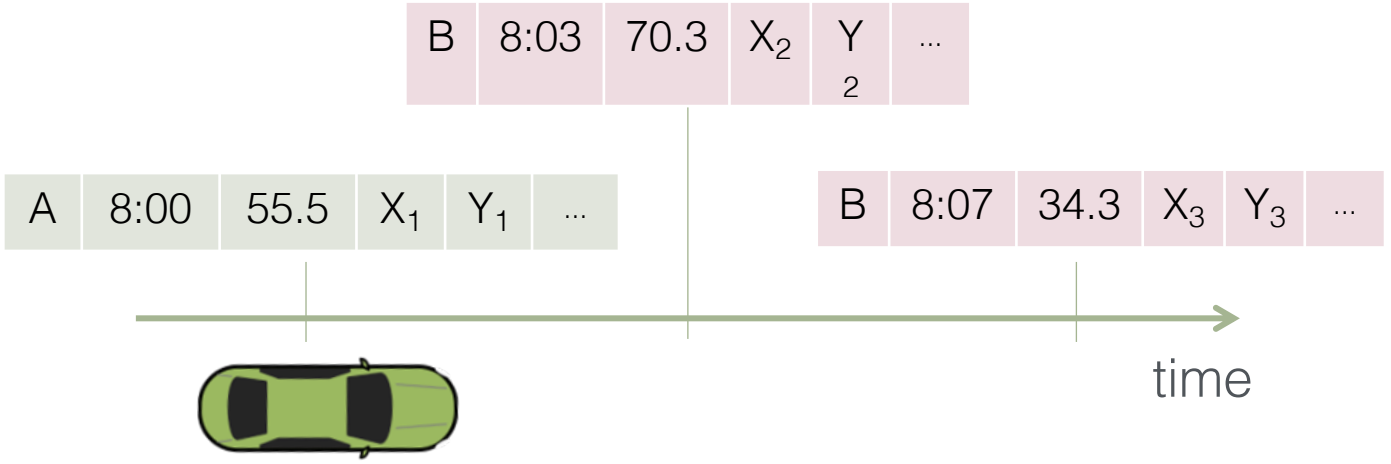
Window Join
Tumbling Window Join
Sliding Window Join
Session Window Join
Interval Join



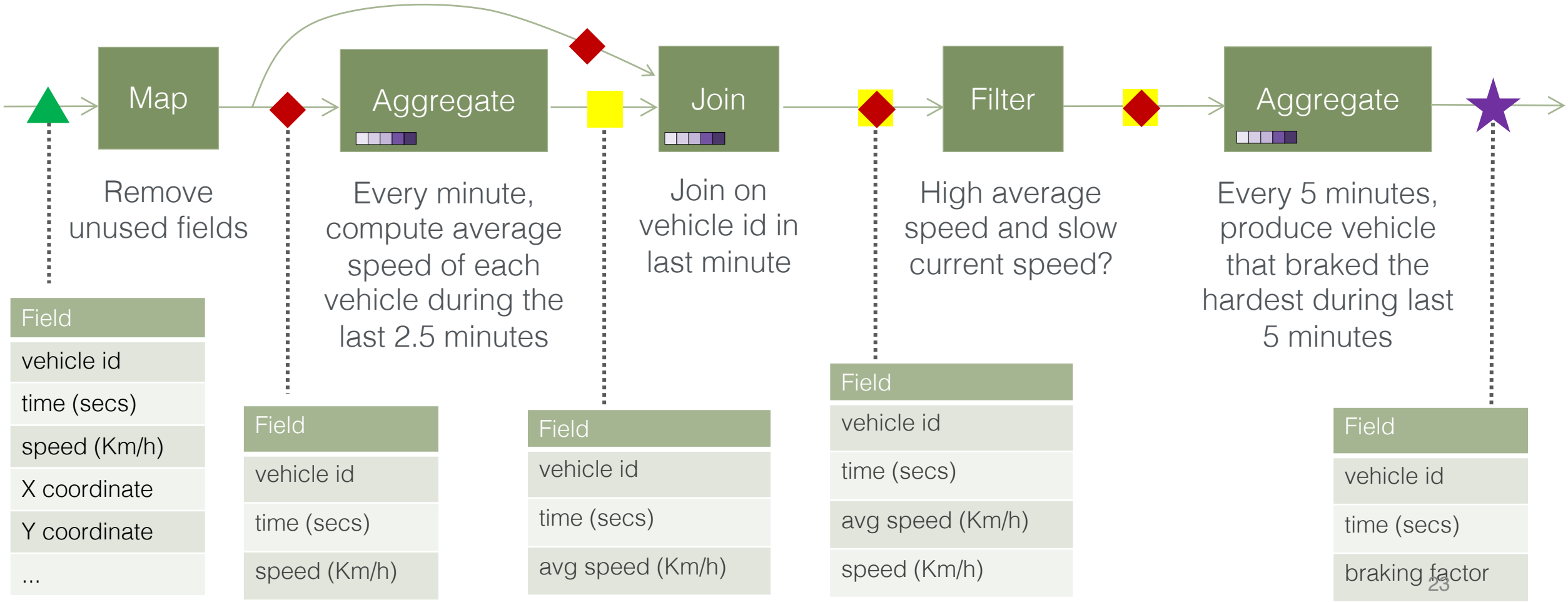
Sample Query

"every five minutes, of all vehicles that braked significantly,
find the one that braked the hardest"





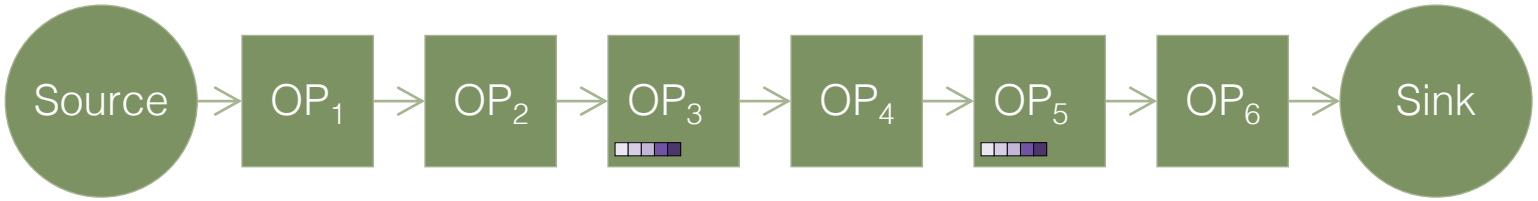
Sample Query

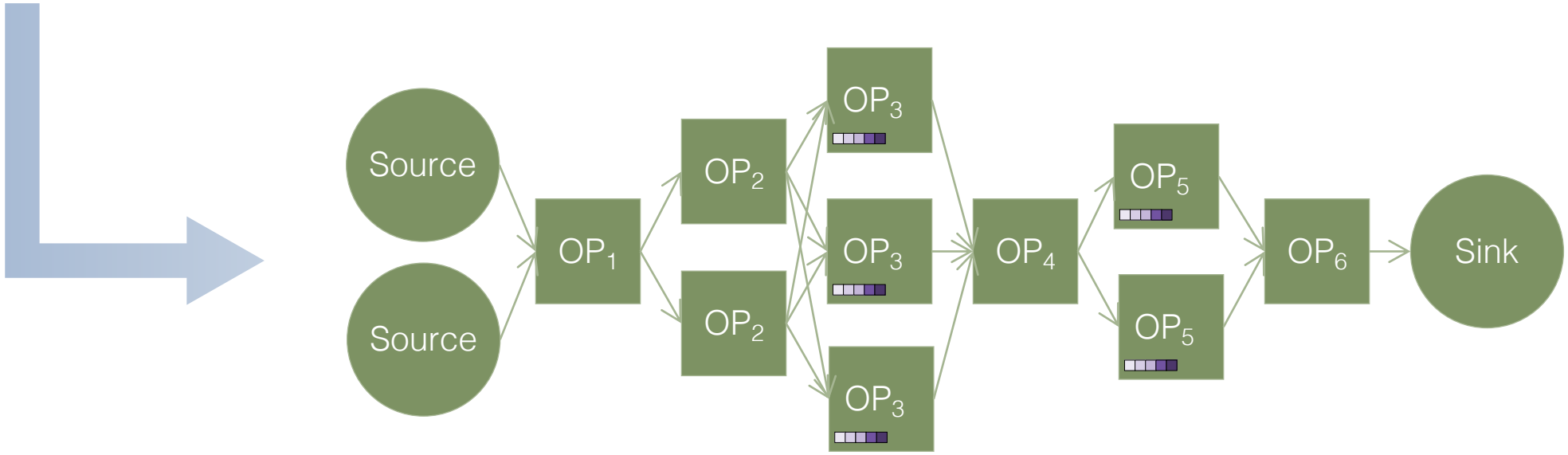


Agenda

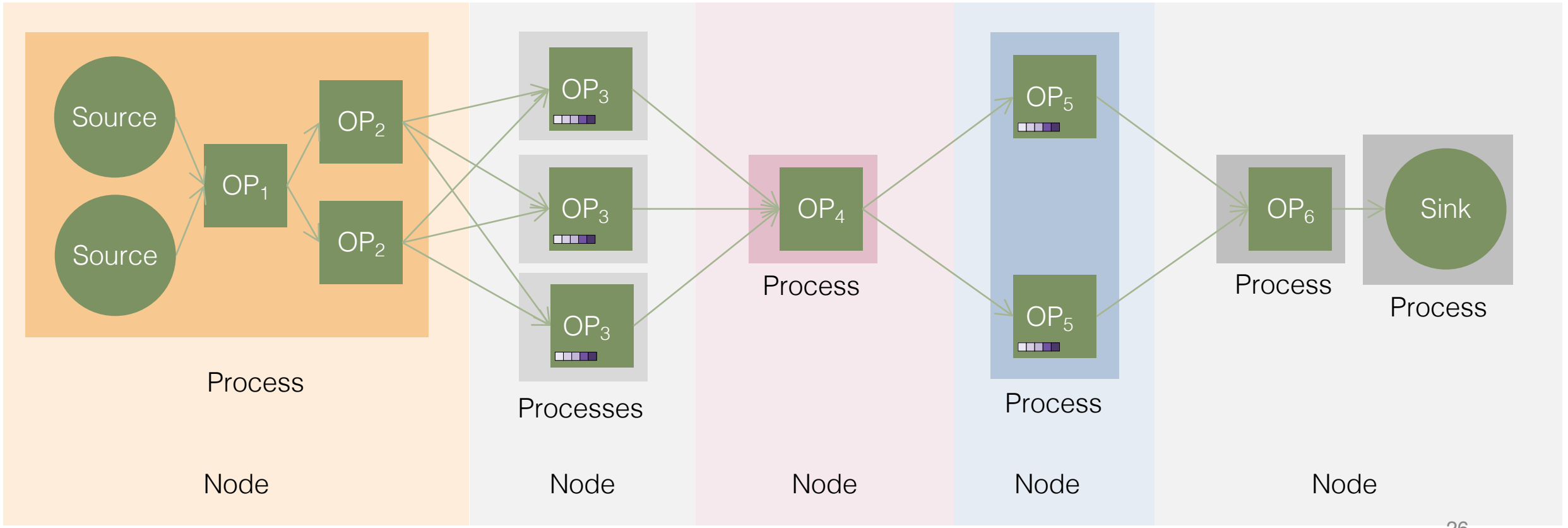
- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

From an abstract query
… to streaming application run by an SPE

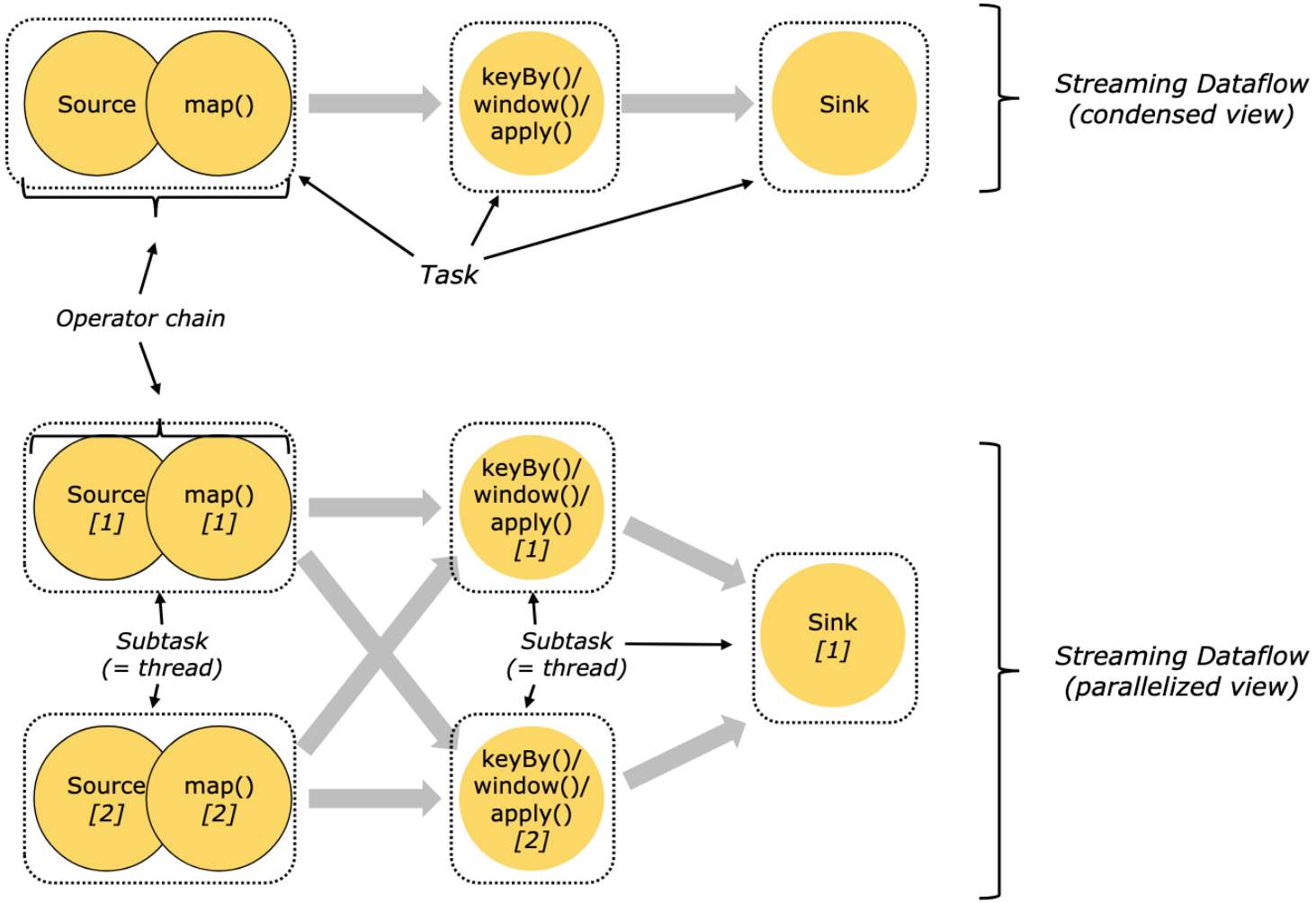




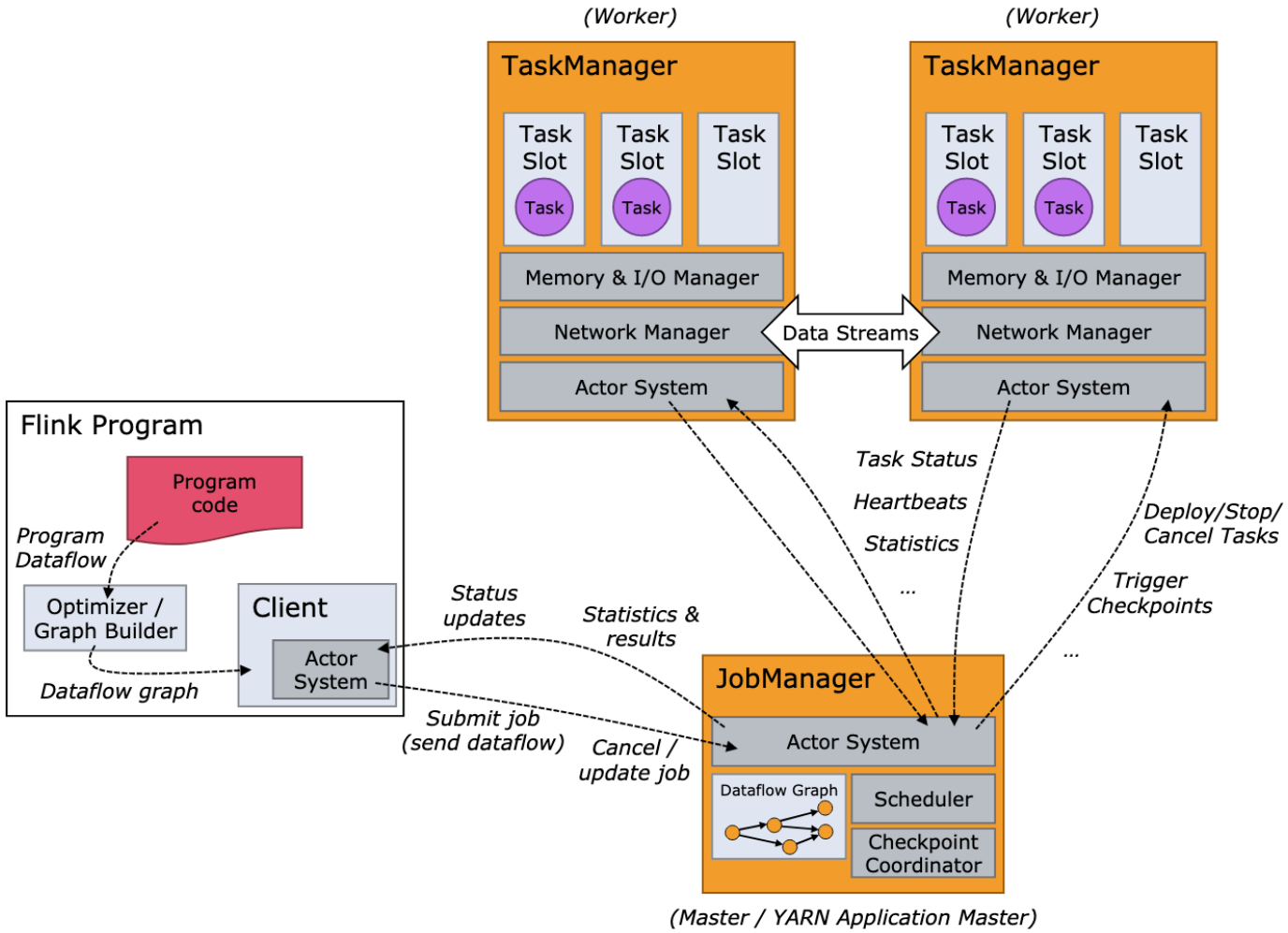
From an abstract query
… to streaming application run by an SPE



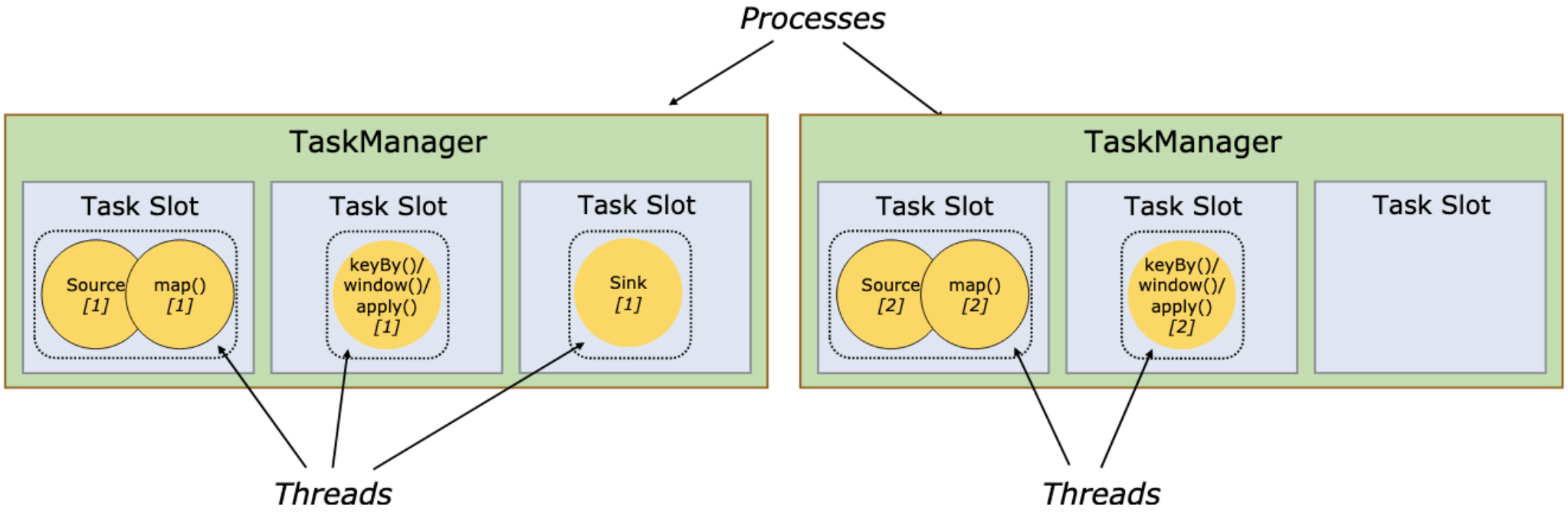
From an abstract query
…to streaming application run by Flink - 1/3



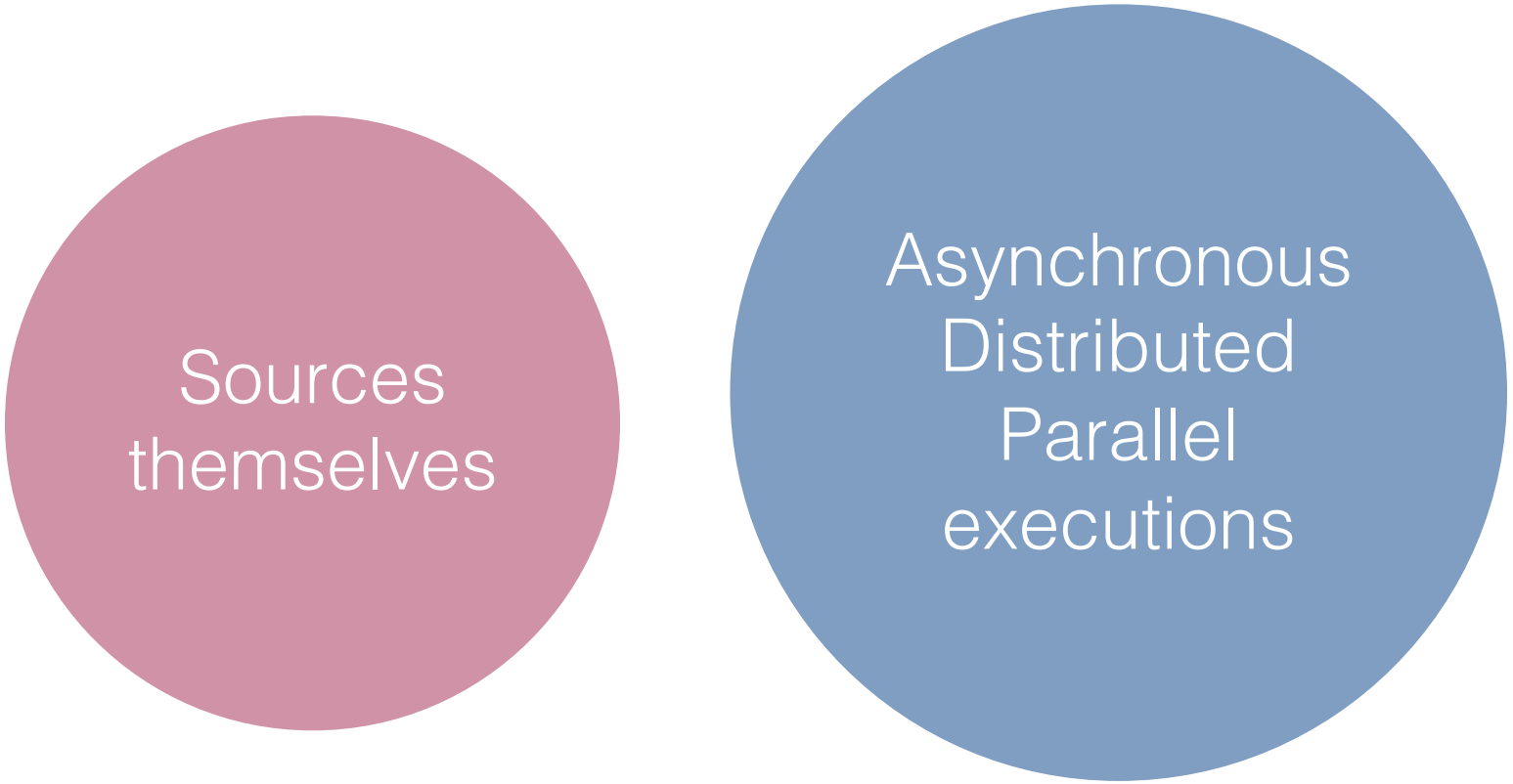
From an abstract query
…to streaming application run by Flink - 1/3



From an abstract query
…to streaming application run by Flink - 1/3



Causes of out-of-order data:



Data sources that produce out-of-order data

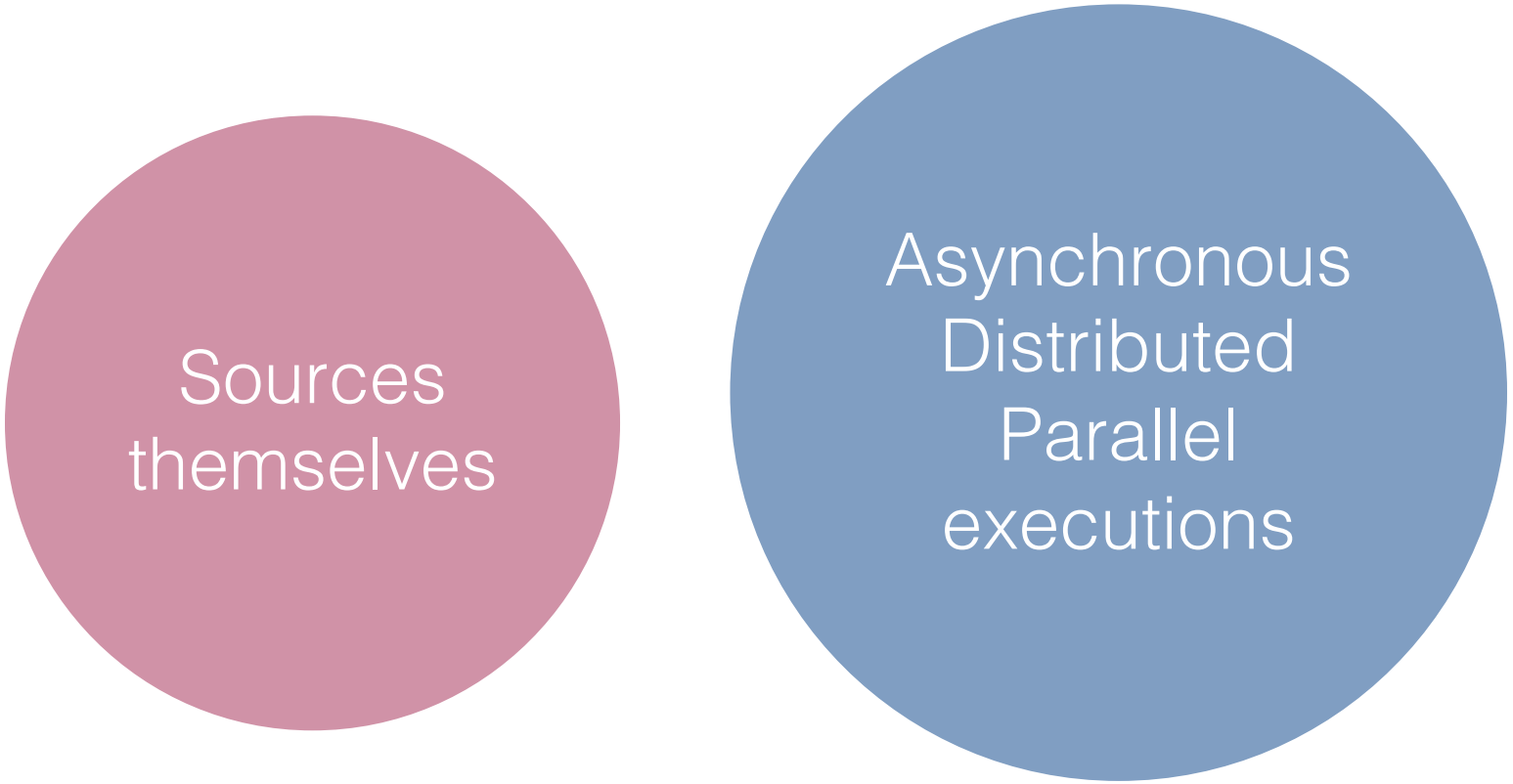
- Discussed in many related works (e.g., Babu, Shivnath, Utkarsh Srivastava, and Jennifer Widom. "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams." ACM Transactions on Database Systems (TODS) 29.3 (2004): 545-580.)
- Battery-operated devices, unreliable wireless networks, …

1 trillion sensors

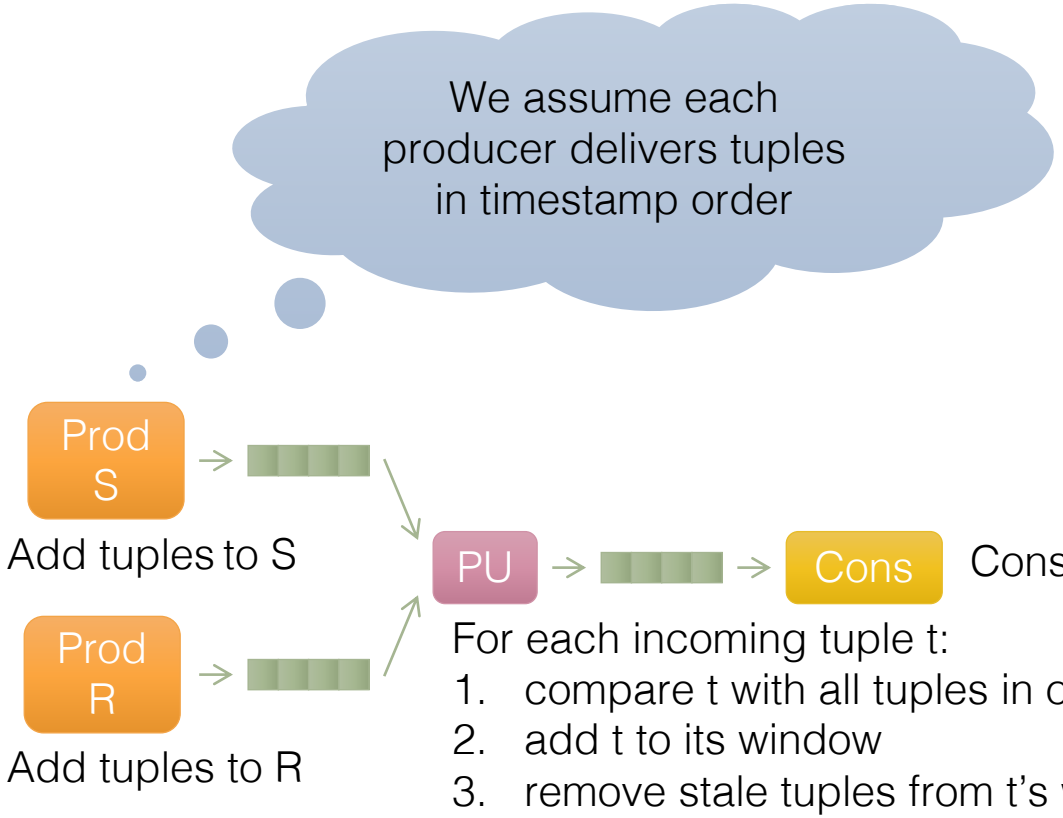
in the **Internet of Things (IoT)**

by 2030

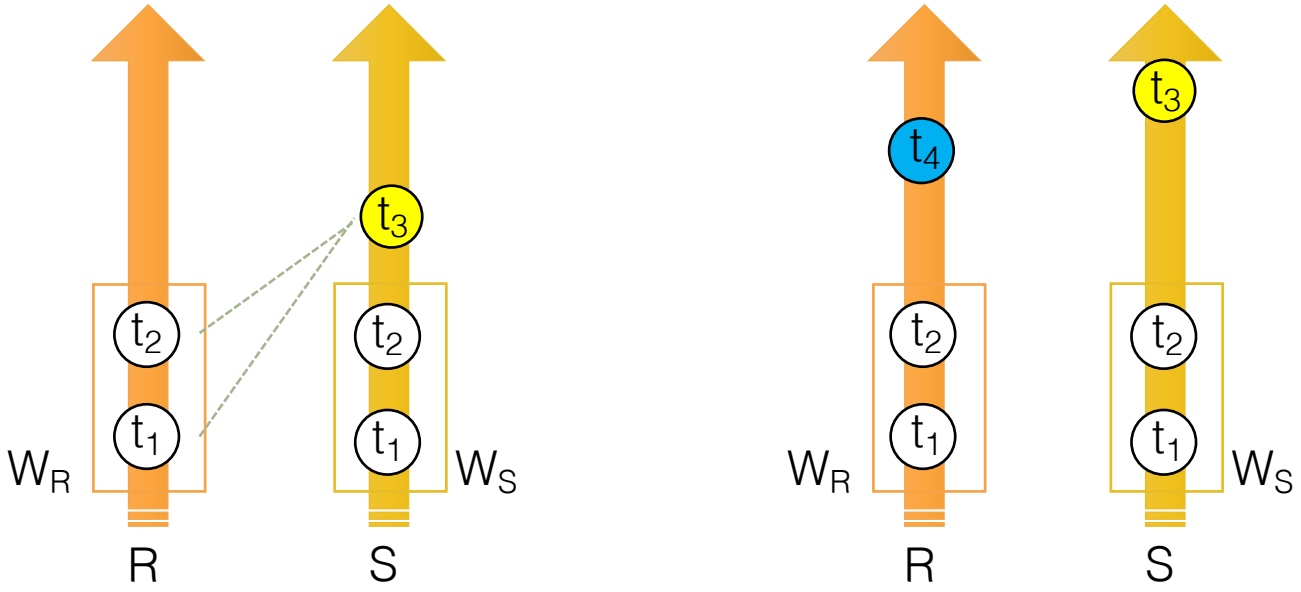
Causes of out-of-order data



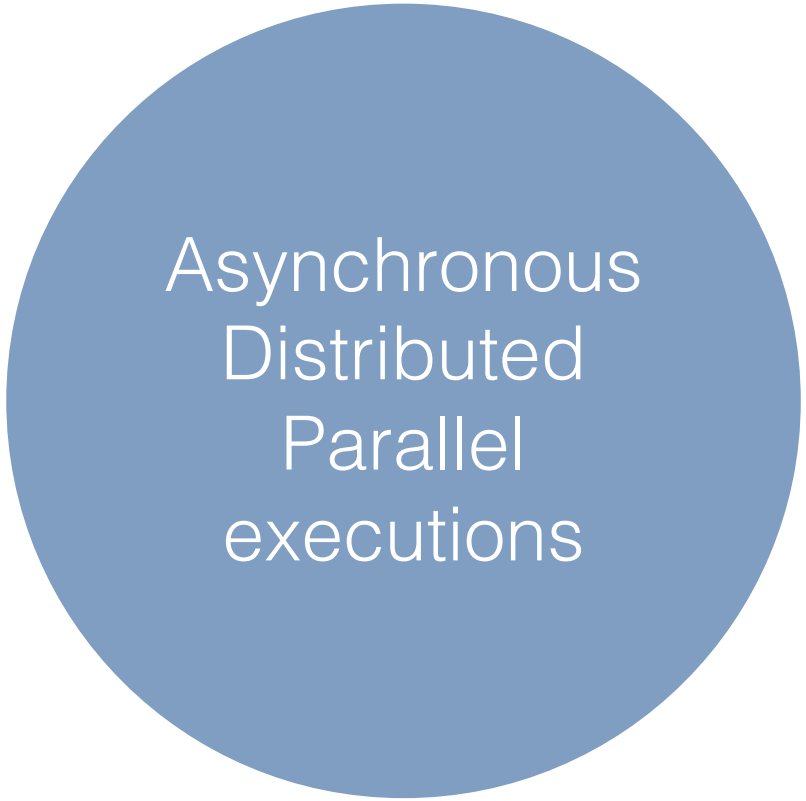
The 3-step procedure (sequential stream join)



The 3-step procedure, is it enough?



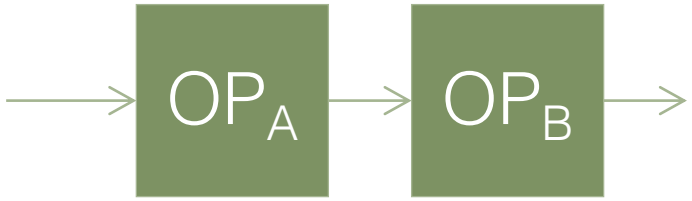
Causes of out-of-order data



Any operator fed data from multiple logical / physical stream can potentially observe out-of-order data

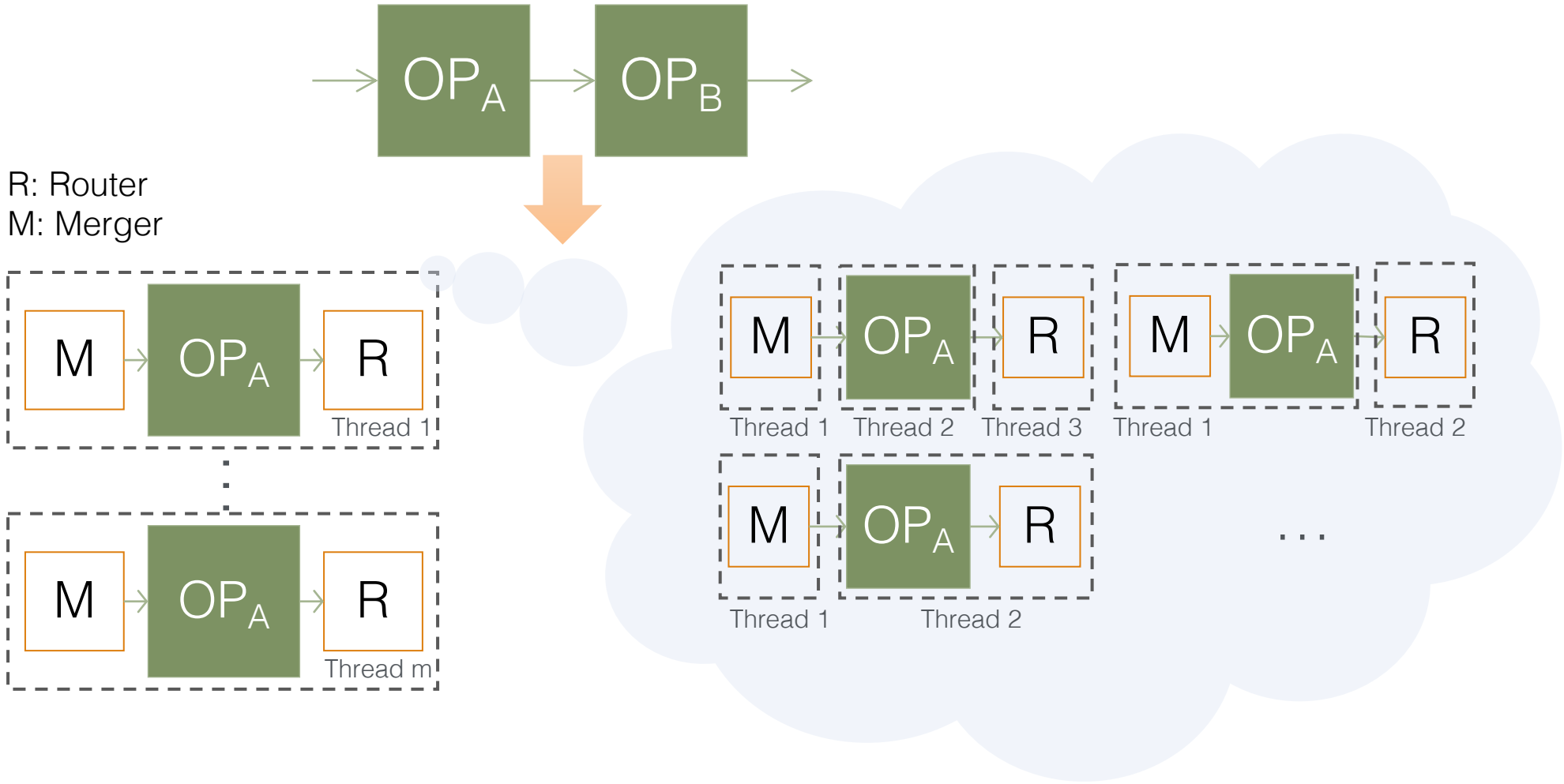
Parallel execution

- General approach



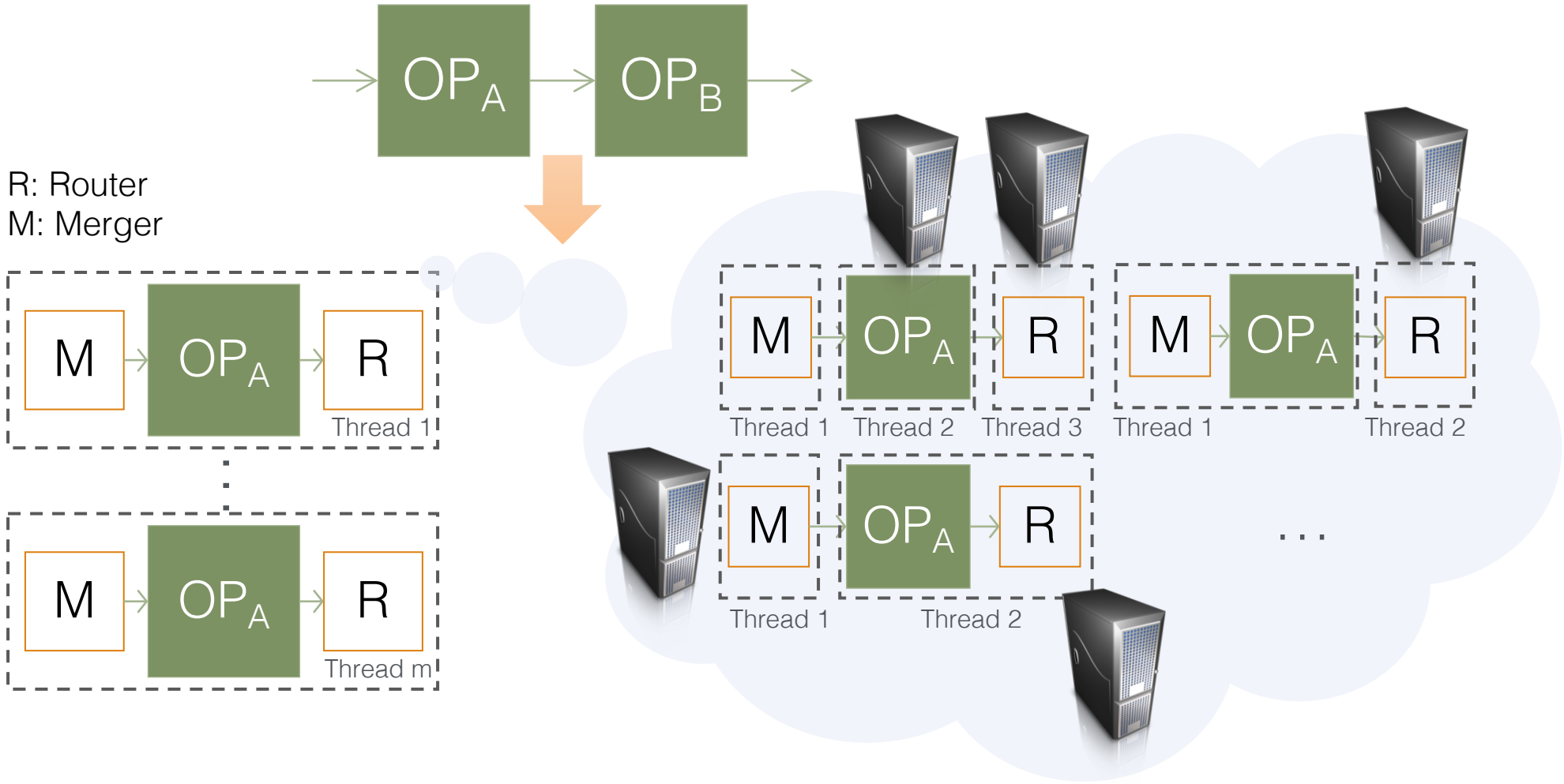
Parallel execution

- General approach



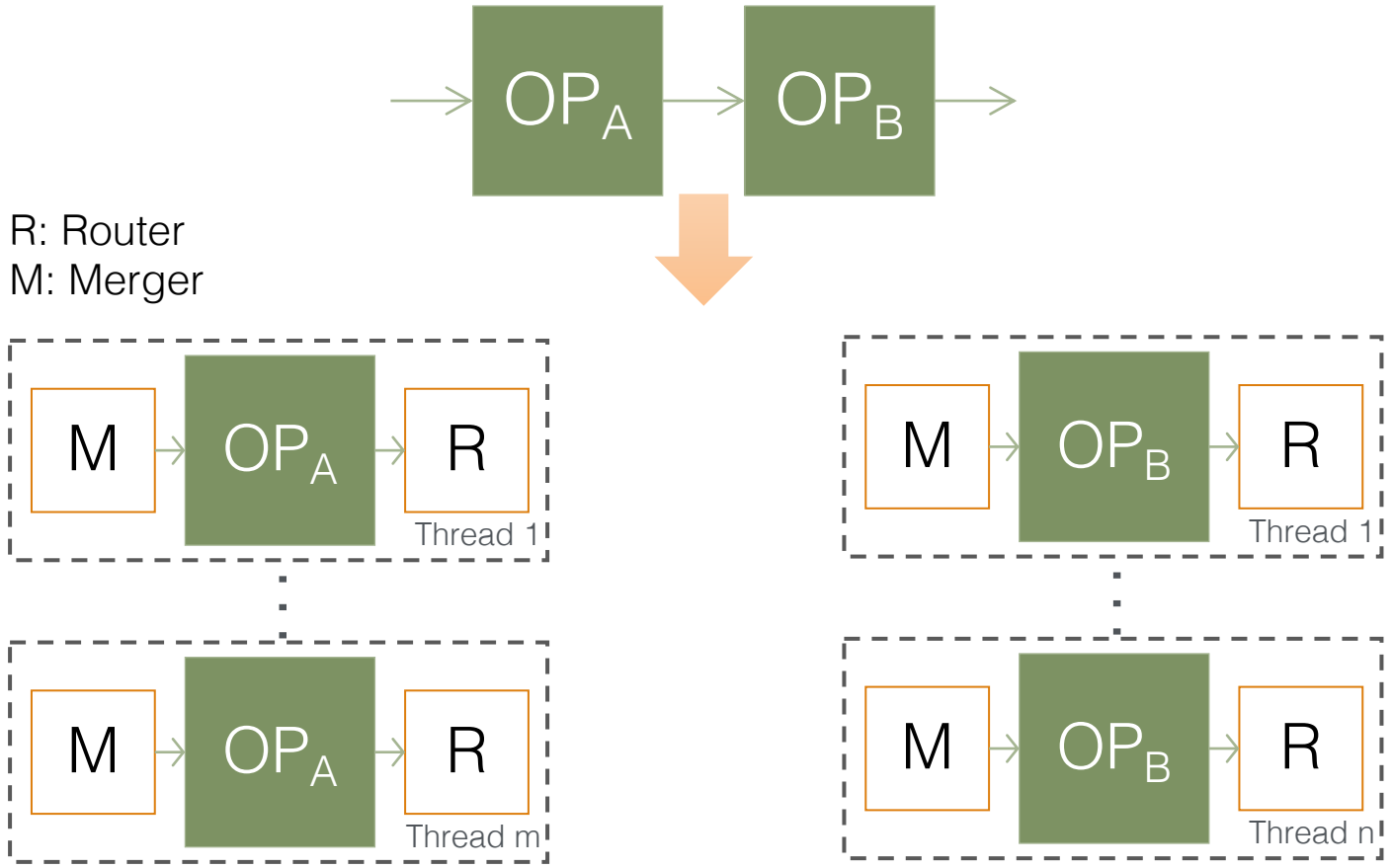
Parallel execution

- General approach



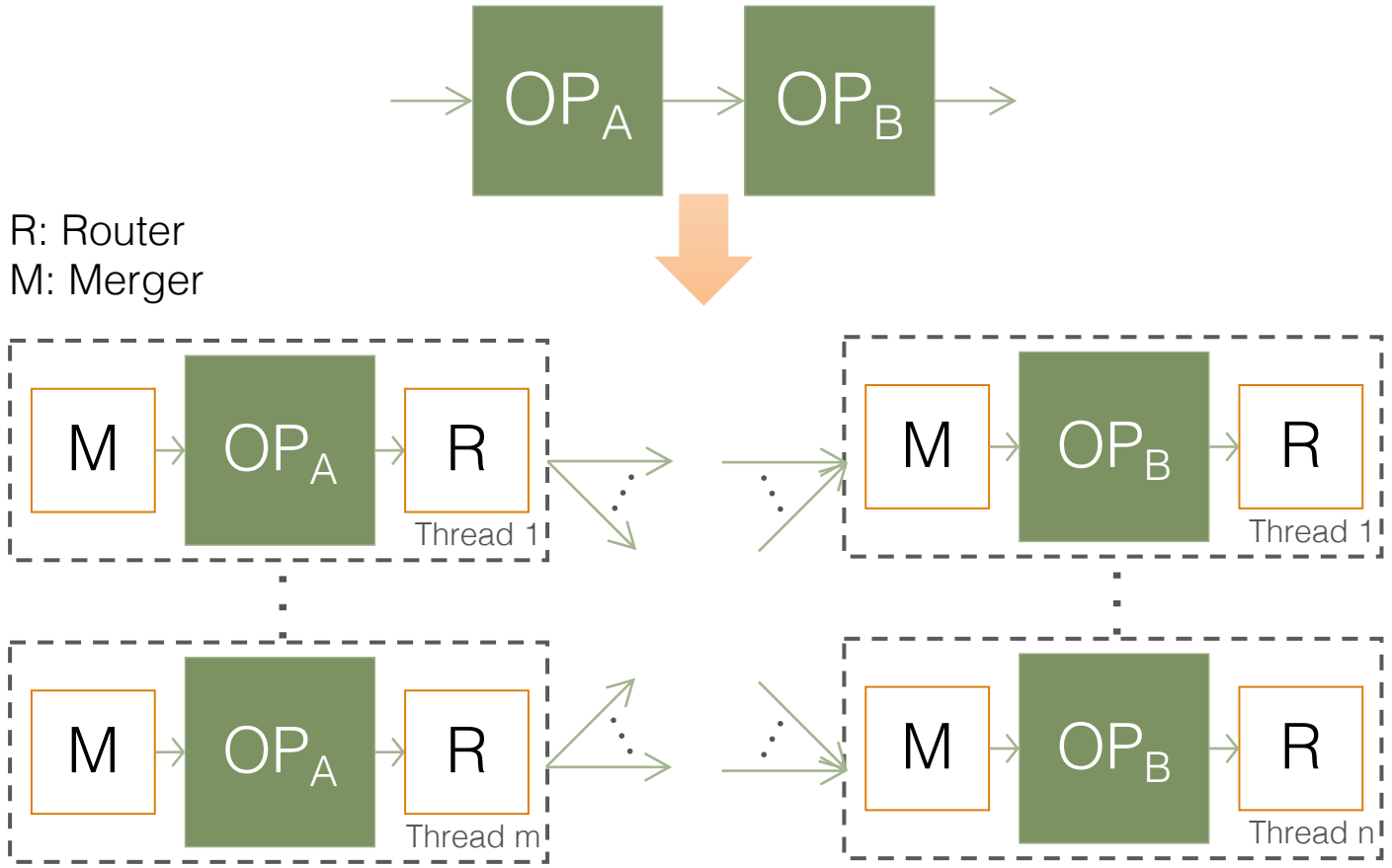
Parallel execution

- General approach



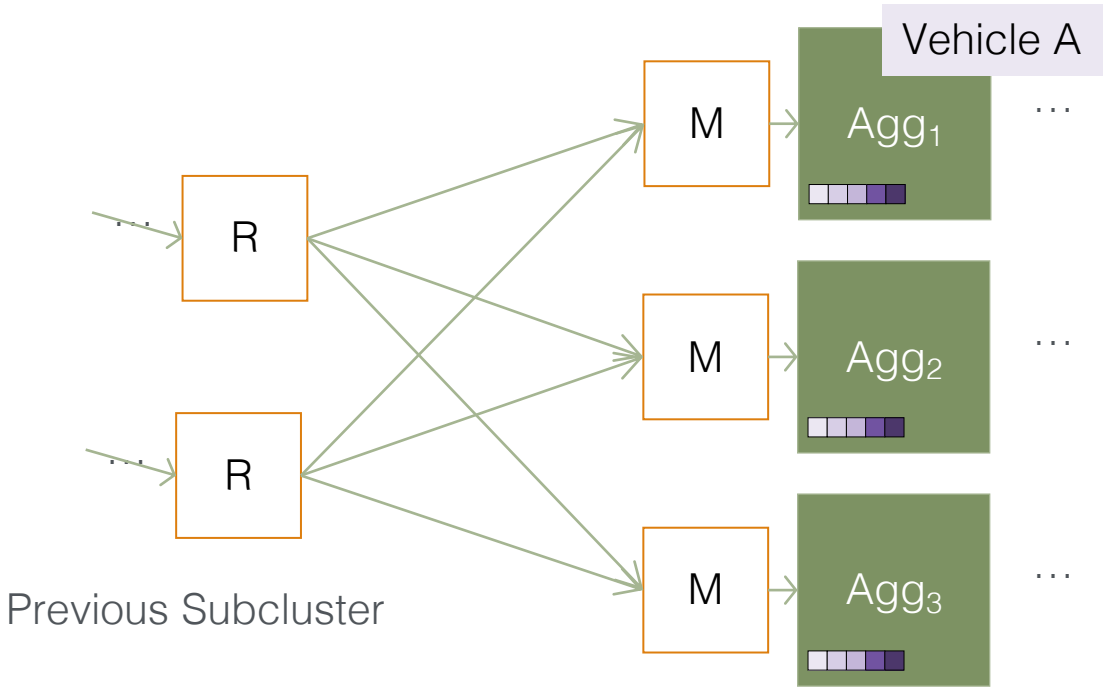
Parallel execution

- General approach



Parallel execution

- Stateful operators: Semantic awareness
  - Aggregate: count within last hour, group-by vehicle id

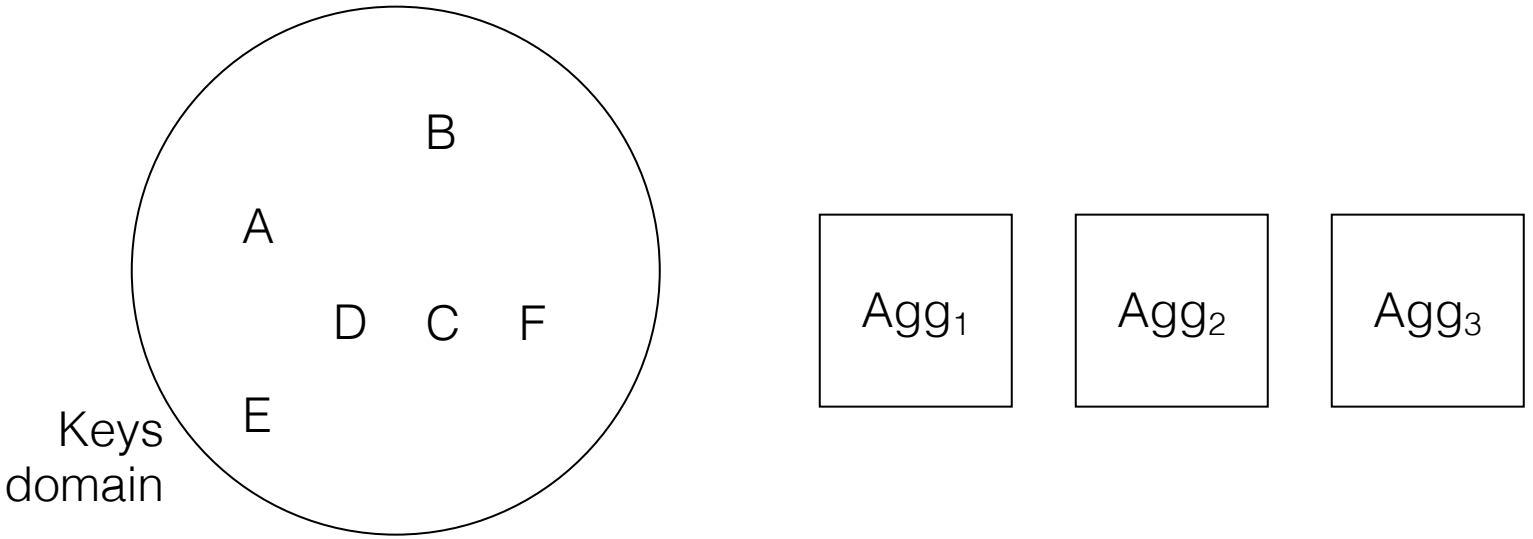


Parallel execution

- Depending on the stateful operator semantic:
  - Partition input stream into **keys**
  - Each key is processed by 1 thread
- # keys >> # threads/nodes

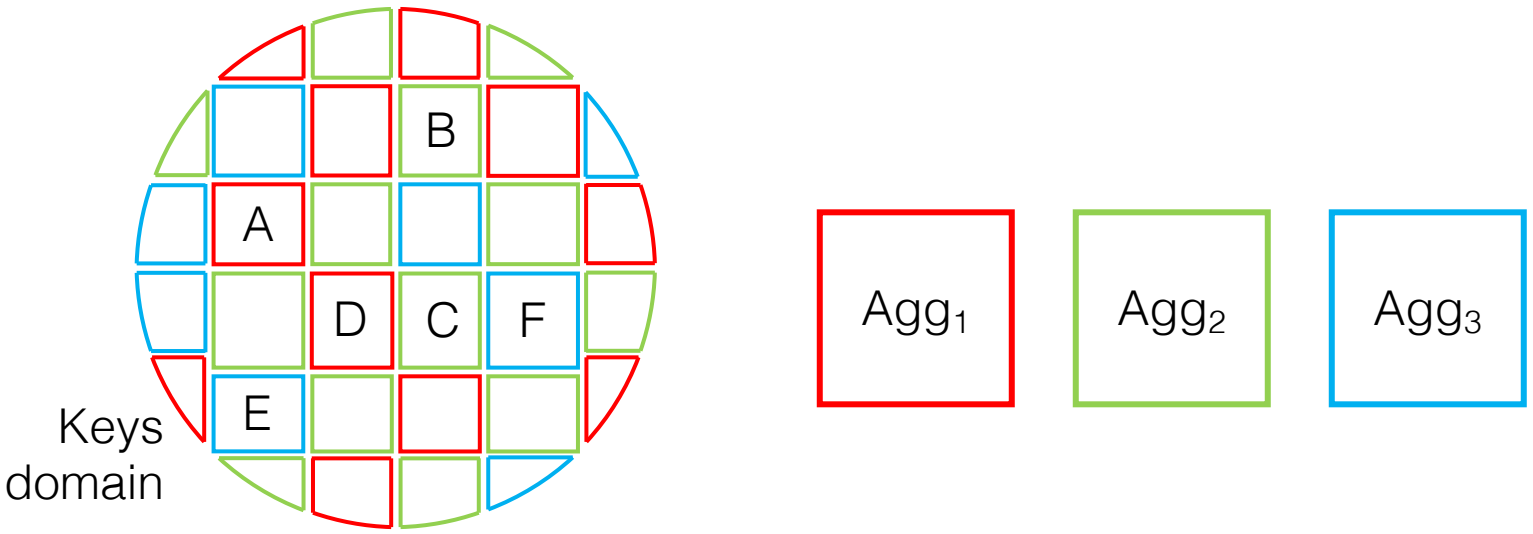
Parallel execution

- Depending on the stateful operator semantic:
  - Partition input stream into **keys**
  - Each key is processed by 1 thread
- # keys >> # threads/nodes



Parallel execution

- Depending on the stateful operator semantics:
  - Partition input stream into **keys**
  - Each key is processed by 1 thread
- # keys >> # threads/nodes

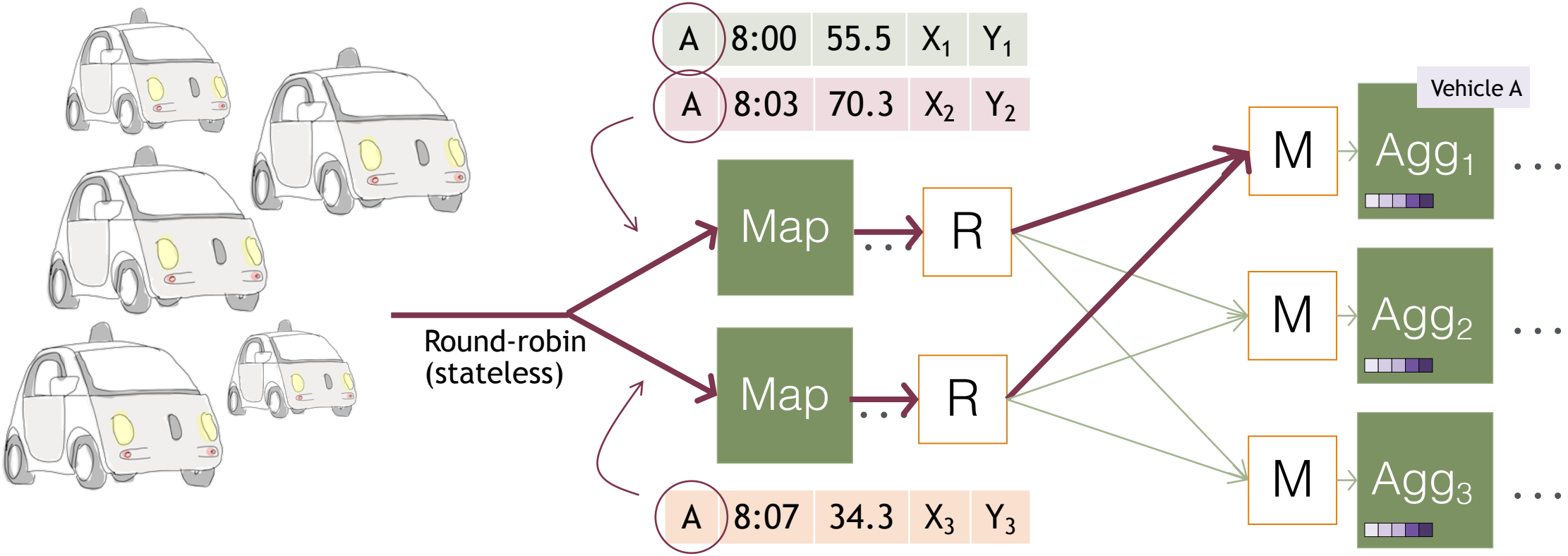


Aggregations
KeyedStream → DataStream
	Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).
	`keyedStream.sum(0);`
	`keyedStream.sum("key");`
	`keyedStream.min(0);`
	`keyedStream.min("key");`
	`keyedStream.max(0);`
	`keyedStream.max("key");`
	`keyedStream.minBy(0);`
	`keyedStream.minBy("key");`
	`keyedStream.maxBy(0);`
	`keyedStream.maxBy("key");`

Window KeyedStream → WindowedStream	Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See windows for a complete description of windows. `dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5))); // Last 5 seconds of data`

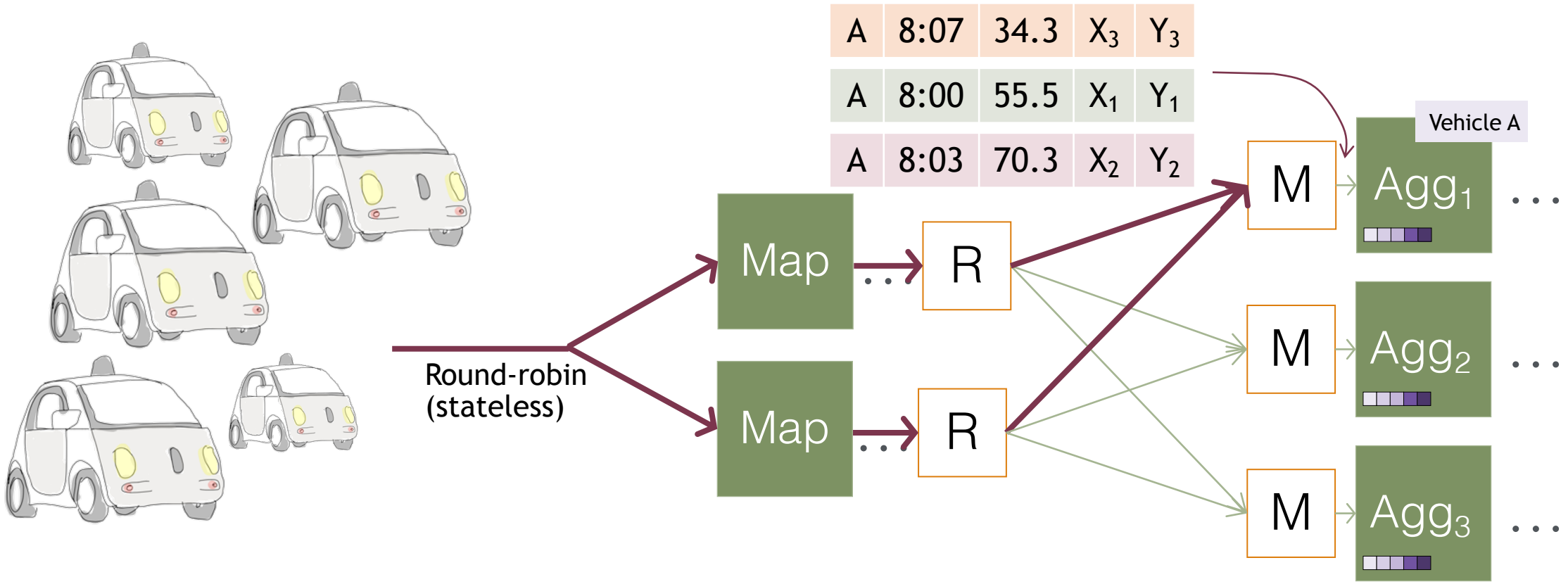
Parallel execution

- Stateful operators: Semantic awareness
  - Aggregate: count within last hour, group-by vehicle id

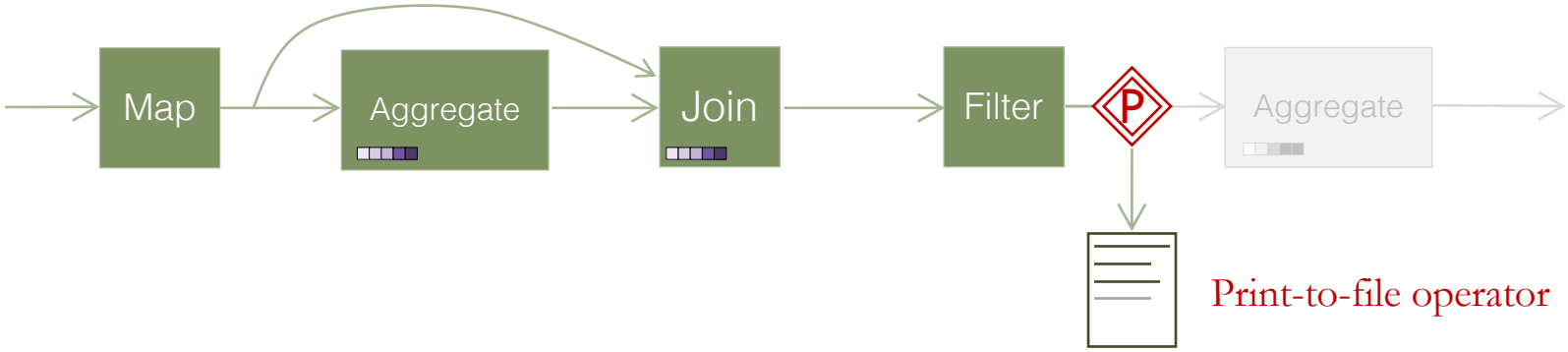


Parallel execution

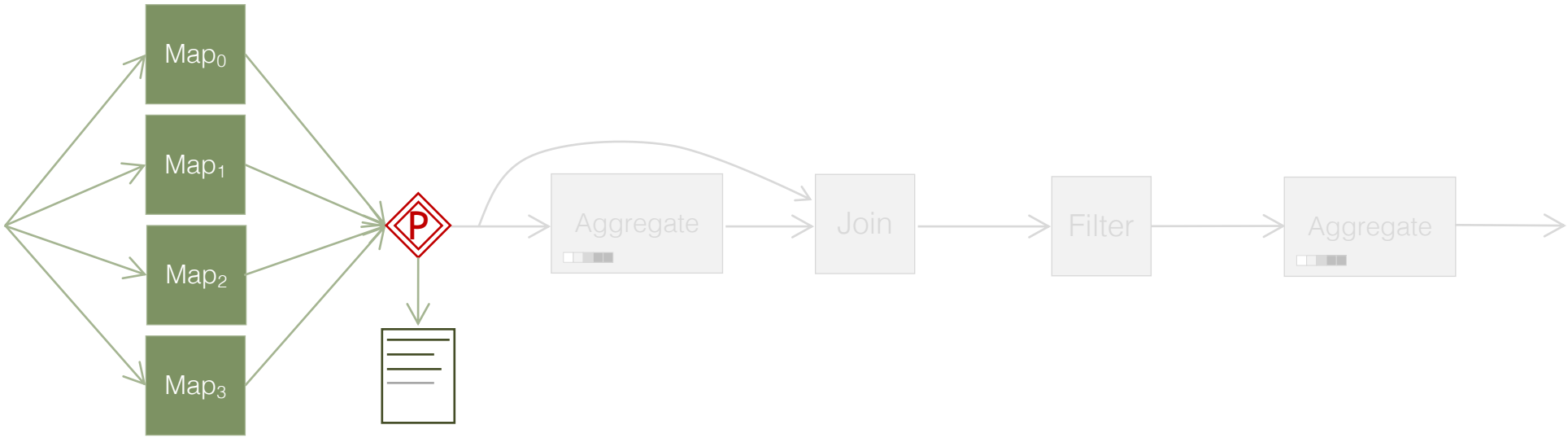
- Stateful operators: Semantic awareness
  - Aggregate: count within last hour, group-by vehicle id



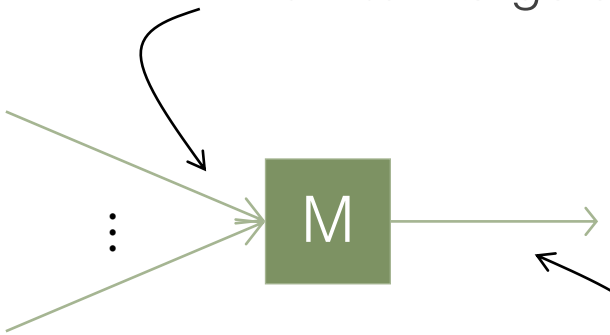
Inherent disorder



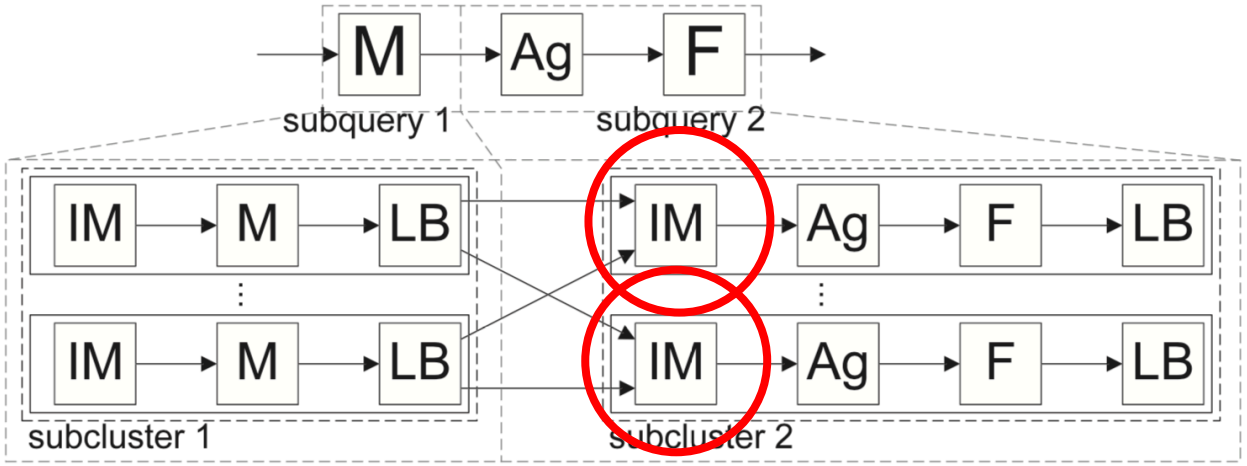
Disorder from parallelism



how to merge several timestamp-sorted streams...



...into one timestamp-sorted stream?

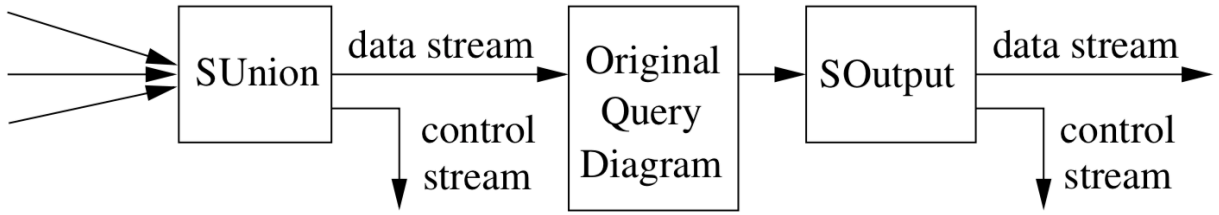


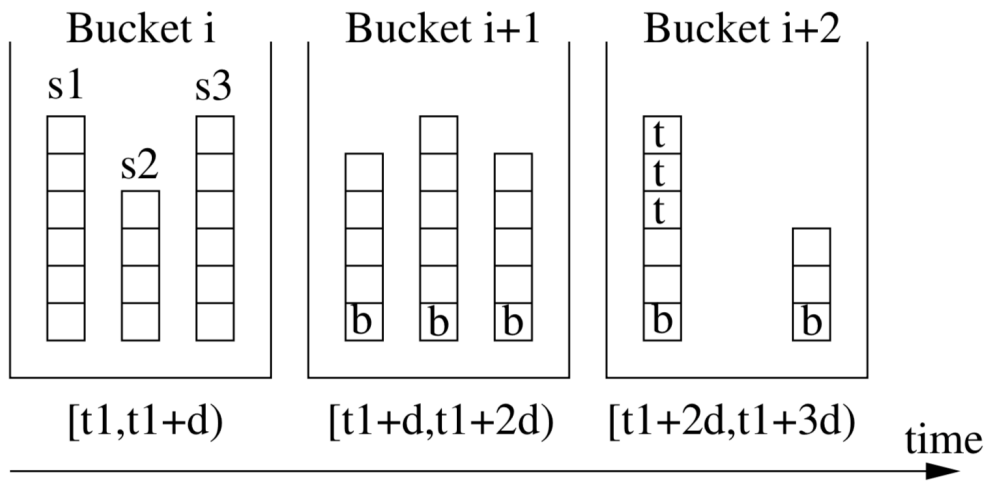
Input Merger

Upon: arrival of t from stream i:

```
9:  buffer[i].enqueue(t)
10: if ∀i, buffer[i].nonEmpty() then
11:     t_o = earliestTuple(buffer)
12:     if ¬ isDummy(t_o) then
13:         forward(t_o)
```

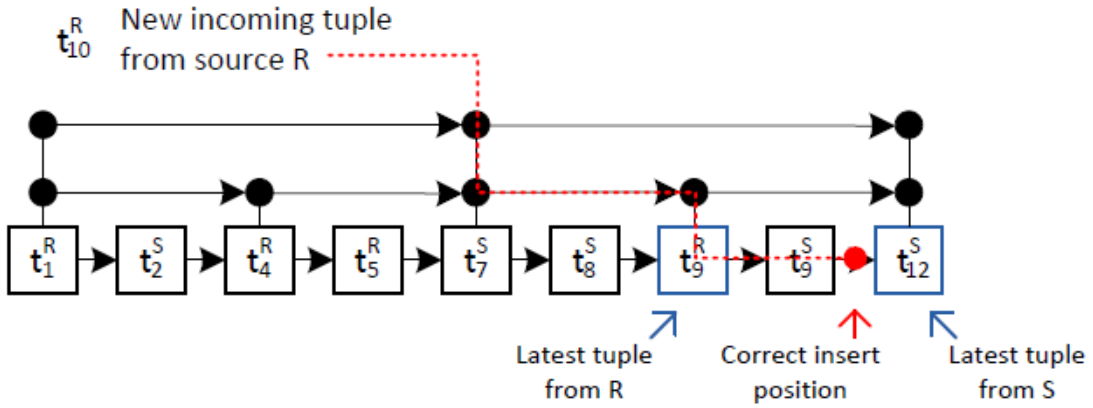
Gulisano, Vincenzo, et al. "Streamcloud: An elastic and scalable data streaming system." IEEE Transactions on Parallel and Distributed Systems 23.12 (2012): 2351-2365.



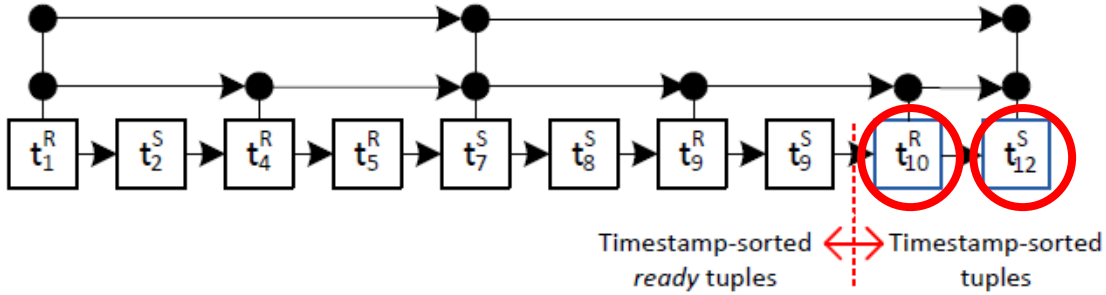


Balazinska, Magdalena, et al. "Fault-tolerance in the Borealis distributed stream processing system." Proceedings of the 2005 ACM SIGMOD international conference on Management of data. 2005.

a) Insertion of a new tuple



b) Distinguishing *ready* from non *ready* tuples



Gulisano, Vincenzo, et al. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join." IEEE Transactions on Big Data (2016).

Which one to choose?

- Different options have different costs
  - might require different types of data structures
  - might require “special tuples” (more on this in the following part of the tutorial)

- When the price of merge-sorting is paid by who provides the tuples rather than who receives them, the system can scale better

Tutorial:

The Role of Event-Time Order in Data Streaming Analysis

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Bastian Havers
Chalmers University of Technology & Volvo Cars
Gothenburg, Sweden
havers@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

Tutorial:

The Role of Event-Time Order in Data Streaming Analysis

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Bastian Havers
Chalmers University of Technology & Volvo Cars
Gothenburg, Sweden
havers@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

Agenda

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

Agenda

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

Pros/Cons of total-ordering

- Cons:
  - Expensive (computation- and latency-wise)
  - An “overkill” for certain applications (more of this in the following slides)

- Pros:
  - Determinism
  - Synchronization
  - Eager purging of stale state

Pros/Cons of total-ordering

- Cons:
  - Expensive (computation- and latency-wise)
  - An “overkill” for certain applications (more of this in the following slides)

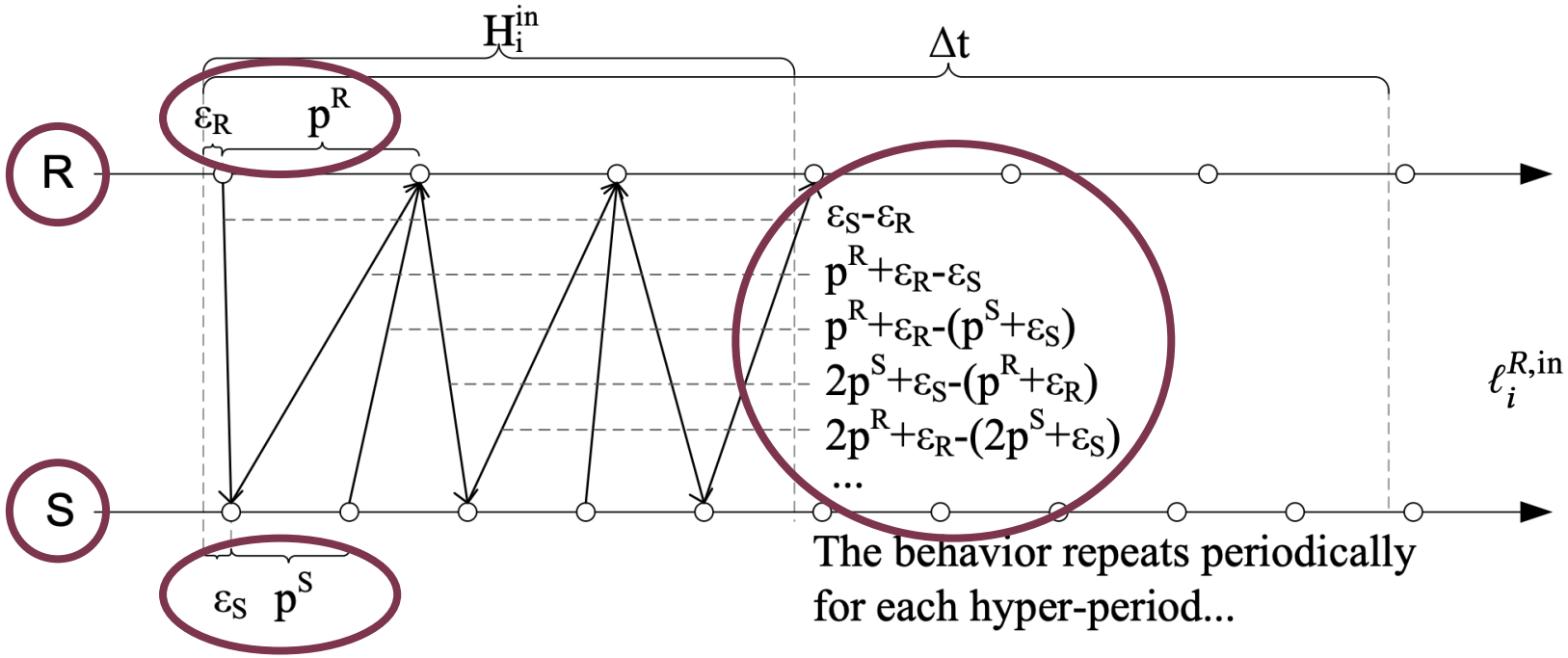
- Pros:
  - Determinism
  - Synchronization
  - Eager purging of stale state

Cost

- We need to temporary maintain tuples
  - Linear in the number of tuples we receive, which depends on the streams' rate
- We need to sort tuples… O(n log(n))
  - (n is number of sources or tuples, depending on the case)
- We need data from all sources, the processing latency depends on the slowest source.
  - The latency overhead can be estimated based on the sources' rates[1]

[1] *Gulisano, Vincenzo, et al. "Performance modeling of stream joins." Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. 2017.*

Estimating the latency overhead



$$H_i^{\text{in}} = \text{LCM}\left(p_i^R, p_i^S\right)$$

$$\ell_i^{R,\text{in}} = \sum_{m=0}^{H_i^{\text{in}} r_i - 1} \left(p_i^S \left\lceil \frac{m p_i^R + \varepsilon_R}{p_i^S} \right\rceil + \varepsilon_S - \left(m p_i^R + \varepsilon_R \right) \right)$$

$$\ell_i^{\text{in}} = \frac{\ell_i^{R,\text{in}} + \ell_i^{S,\text{in}}}{H_i^{\text{in}} \left(r_i + s_i \right)}.$$

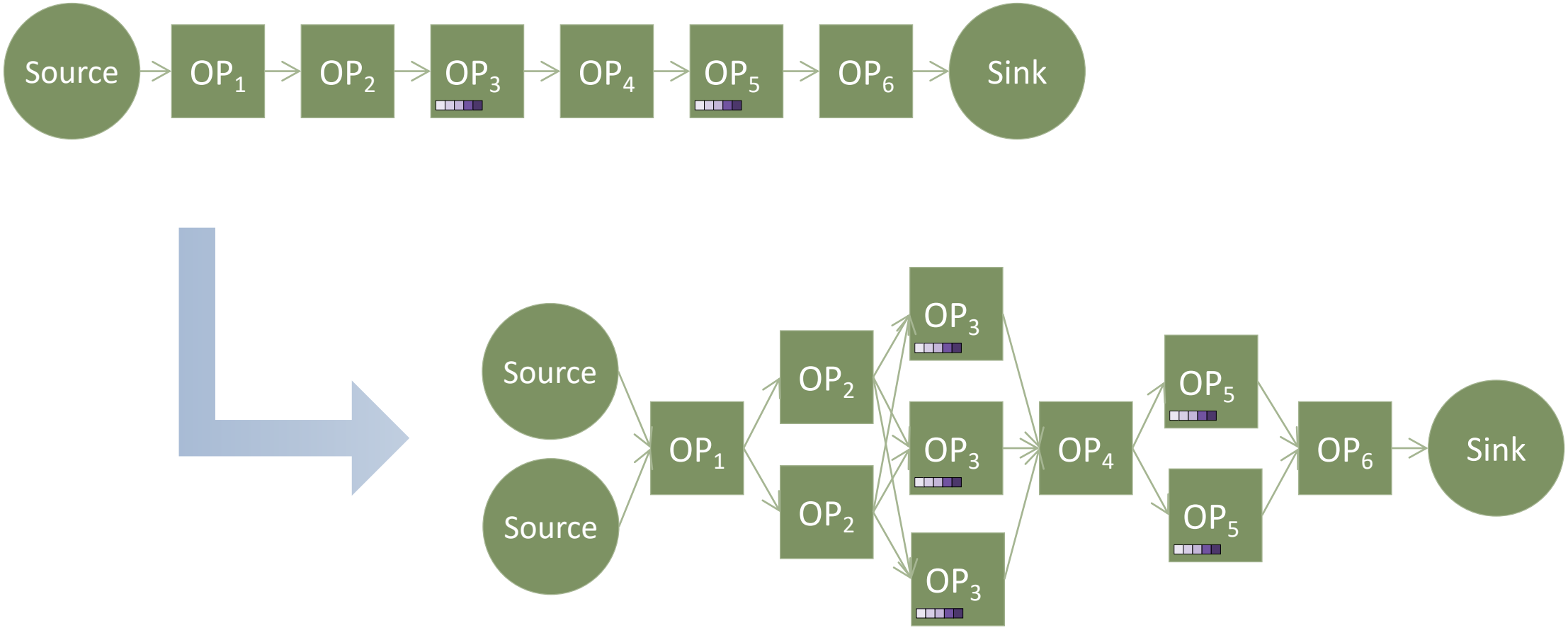
[1]Gulisano, Vincenzo, et al. "Performance modeling of stream joins." Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. 2017.

Pros/Cons of total-ordering

- Cons:
  - Expensive (computation- and latency-wise)
  - An “overkill” for certain applications (more of this in the following slides)

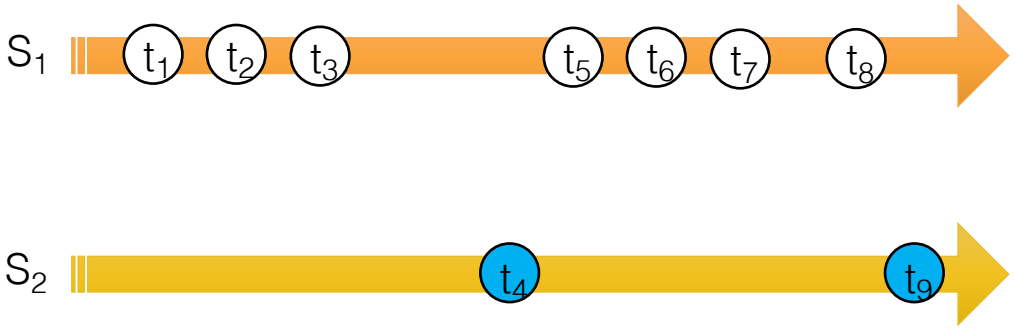
- Pros:
  - Determinism
  - Synchronization
  - Eager purging of stale state

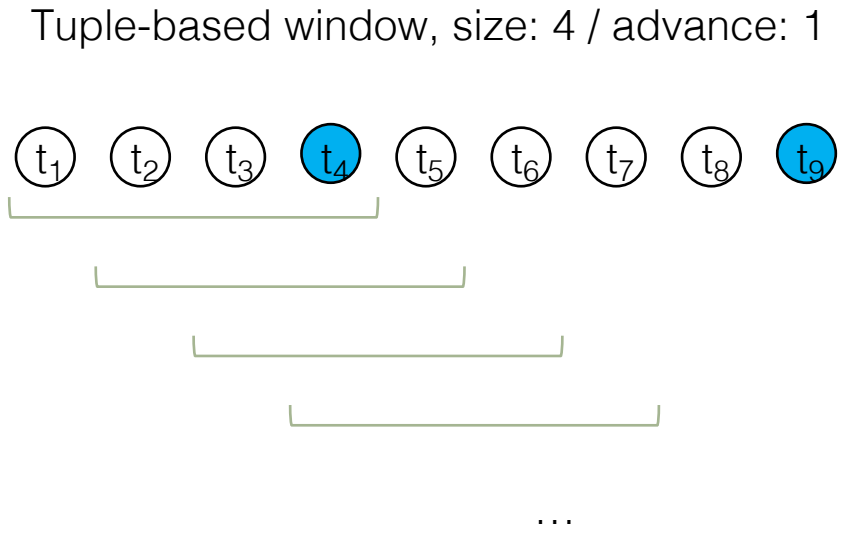
Determinism



Balazinska, Magdalena, et al. "Fault-tolerance in the Borealis distributed stream processing system." Proceedings of the 2005 ACM SIGMOD international conference on Management of data. 2005.
Gulisano, Vincenzo. StreamCloud: an elastic parallel-distributed stream processing engine. Diss. 2012.

Determinism





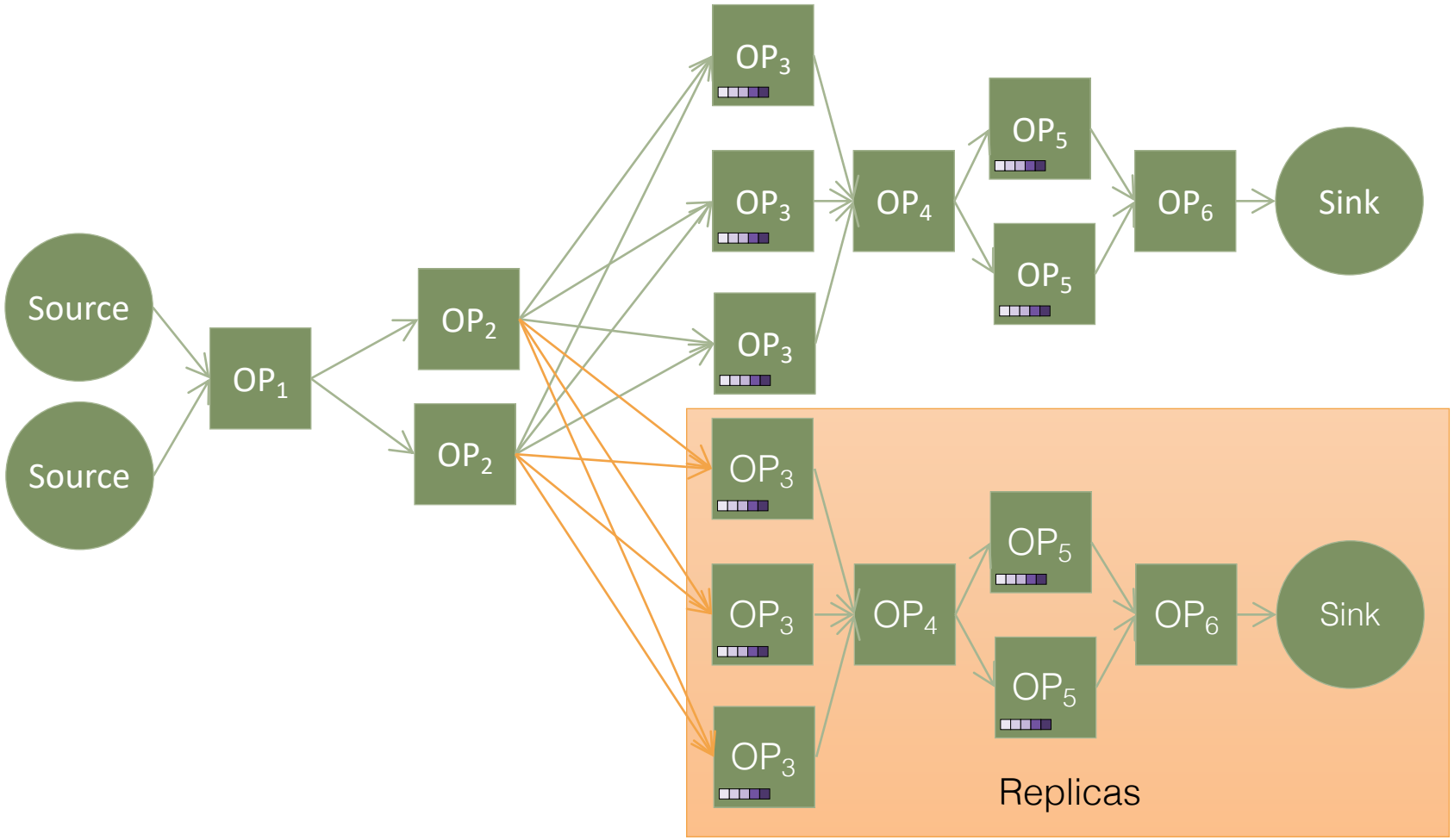
Hwang, Jeong-Hyon, Ugur Cetintemel, and Stan Zdonik. "Fast and reliable stream processing over wide area networks." 2007 IEEE 23rd International Conference on Data Engineering Workshop. IEEE, 2007.
Gulisano, Vincenzo, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join." IEEE Transactions on Big Data (2016).

Pros/Cons of total-ordering

- Cons:
  - Expensive (computation- and latency-wise)
  - An “overkill” for certain applications (more of this in the following slides)

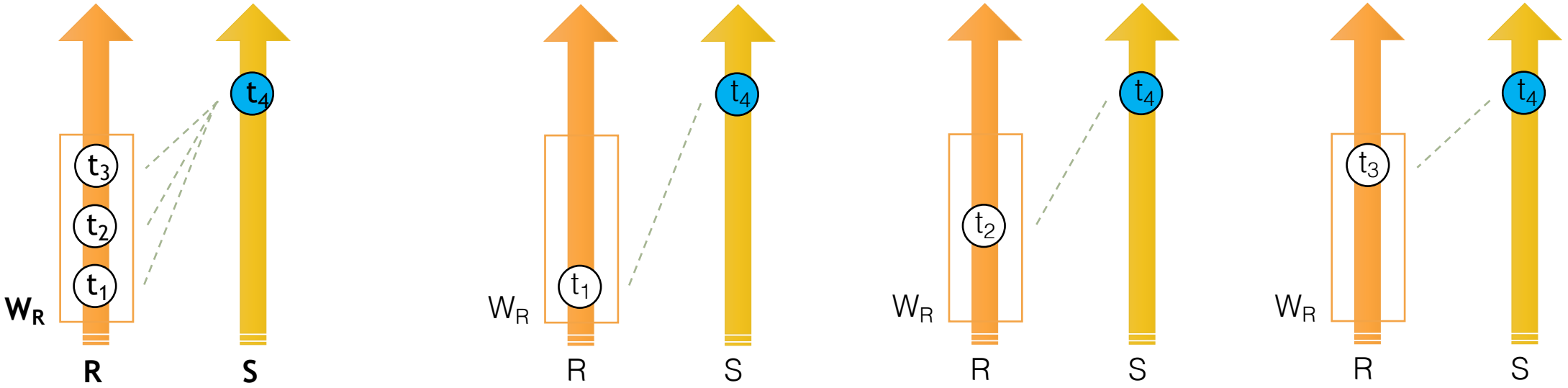
- Pros:
  - Determinism
  - Synchronization
  - Eager purging of stale state

Synchronization



Balazinska, Magdalena, et al. "Fault-tolerance in the Borealis distributed stream processing system." Proceedings of the 2005 ACM SIGMOD international conference on Management of data. 2005.

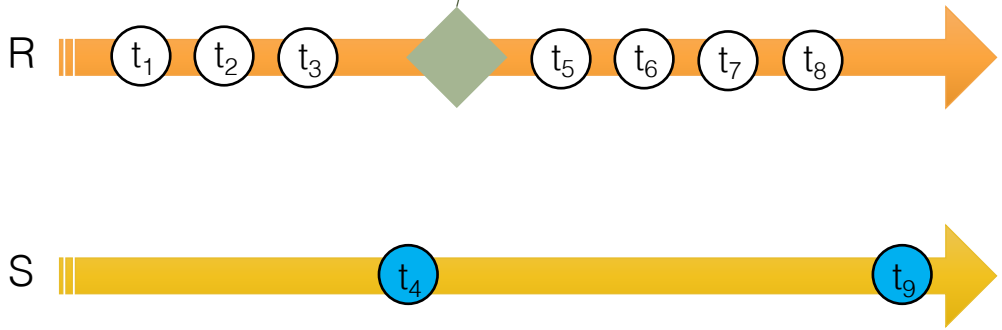
Synchronization

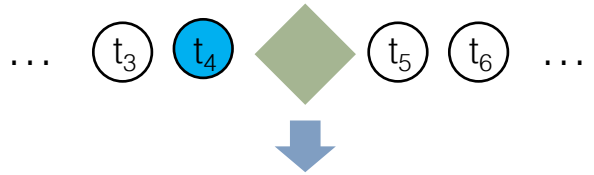


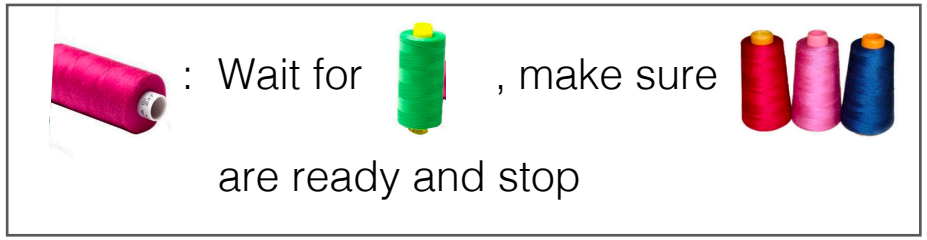
Gulisano, Vincenzo, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join." IEEE Transactions on Big Data (2016).

Synchronization

- Act as a barrier
- Carry the reconfiguration to be applied
  - Change the parallelism degree
  - Change other configuration





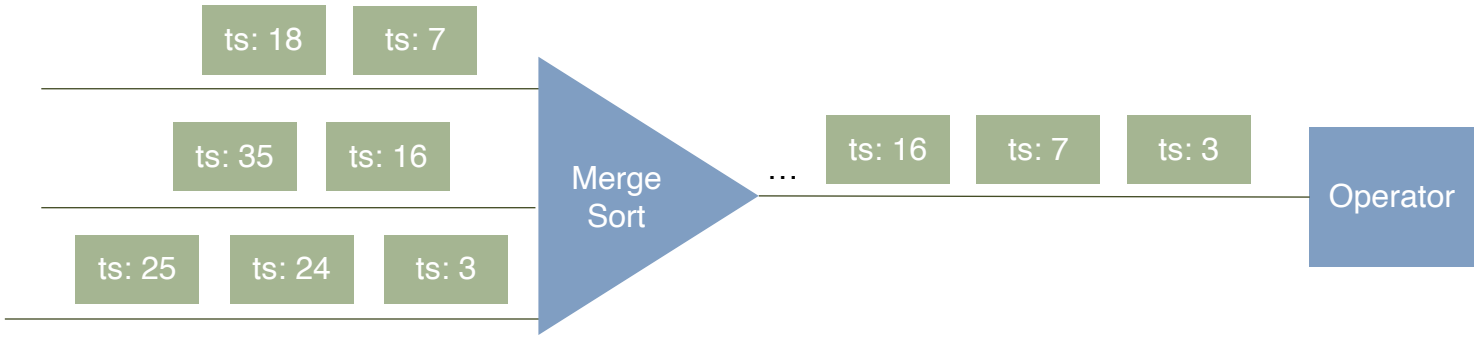


Najdataei, Hannaneh, et al. "Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism." Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems. 2019.

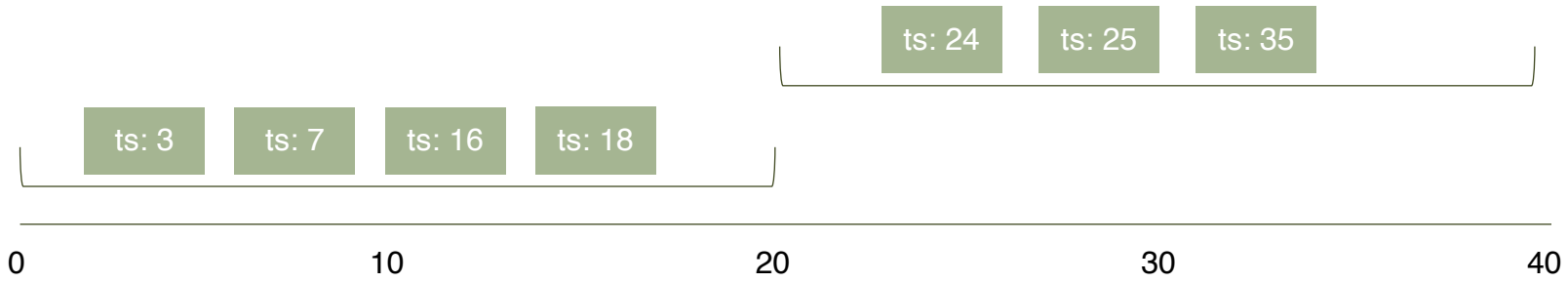
Agenda

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

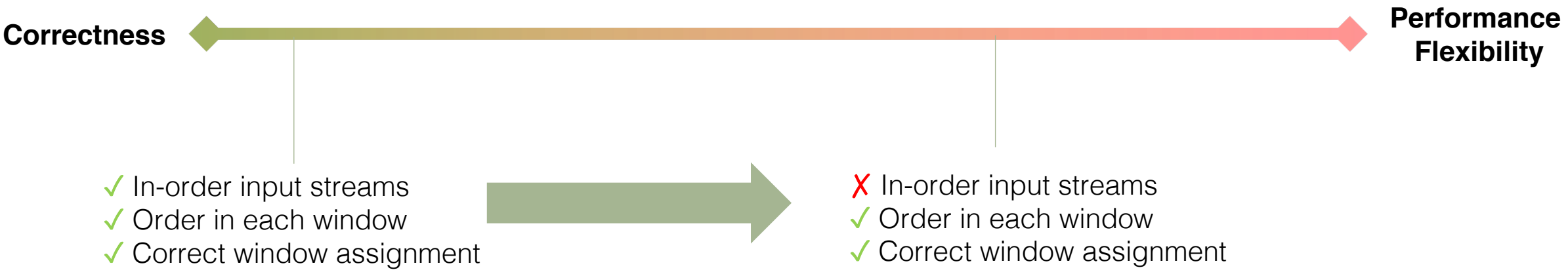
Total-Ordering Recap



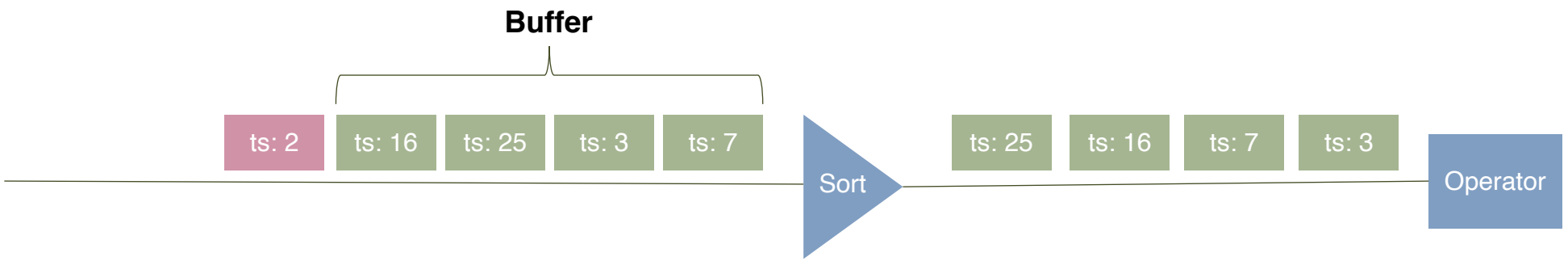
- ✓ Correct window assignment
- ✓ Order in each window



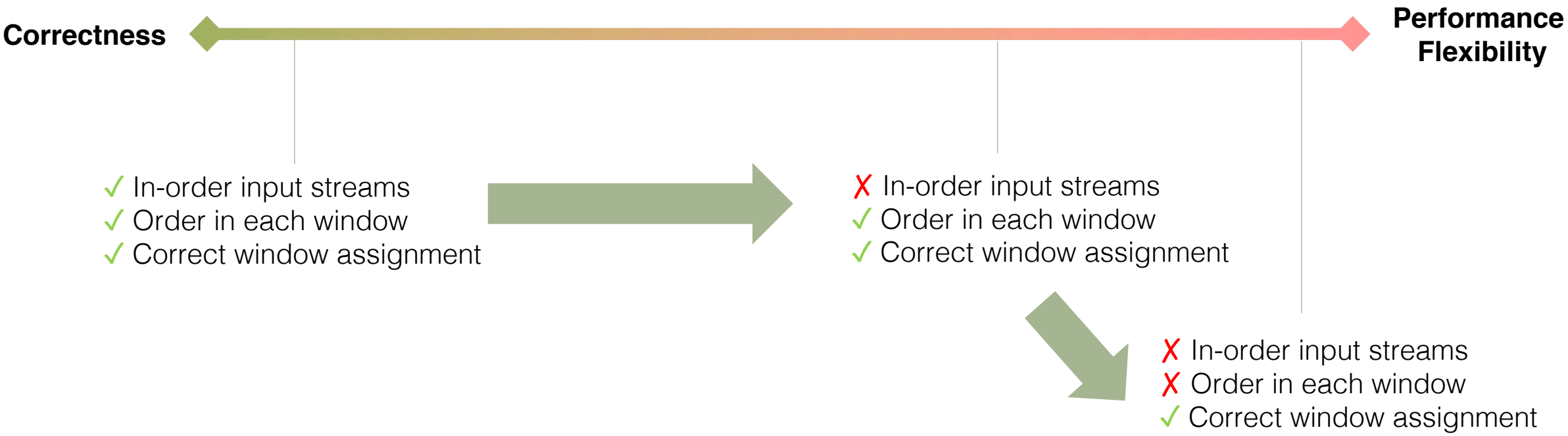
Relaxing Correctness



Buffering Approaches



Relaxing Correctness



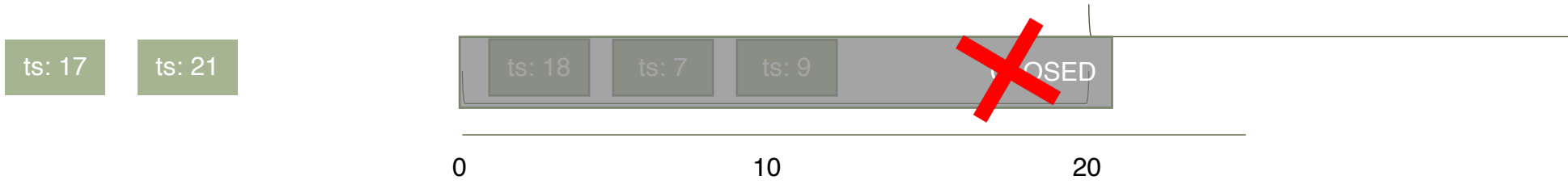
Disorder + Correct Window Assignment

1. When to **create** each window?
2. When to **close** each window (and produce result)?

Creating Windows

ts: 9 ts: 7 ts: 18

Closing Windows



Operator needs some guarantee
it will not receive tuples with $\mathbf{ts} < \mathbf{W}$

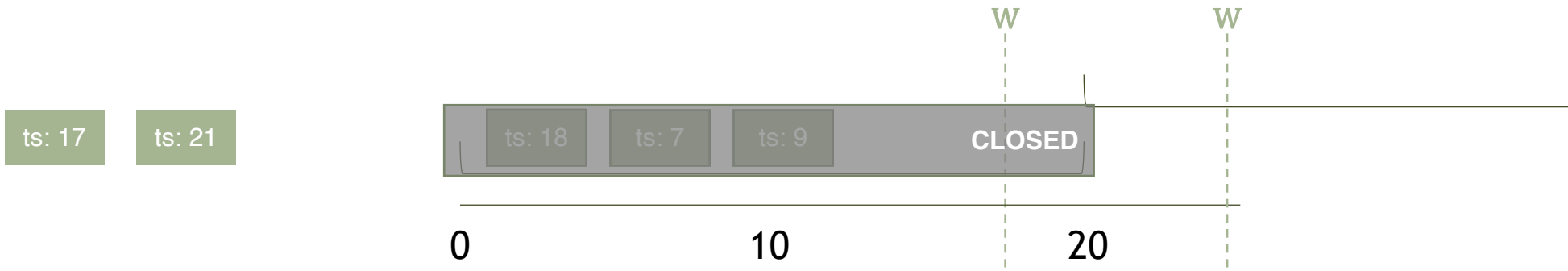
Safely close all windows where
$\mathbf{right\_boundary} \leq \mathbf{W}$

Watermarks

Assume we can compute a monotonic function

$$F: O \rightarrow E$$

that returns $W \in E$ the earliest event time of any tuple that can arrive at operator O in the future



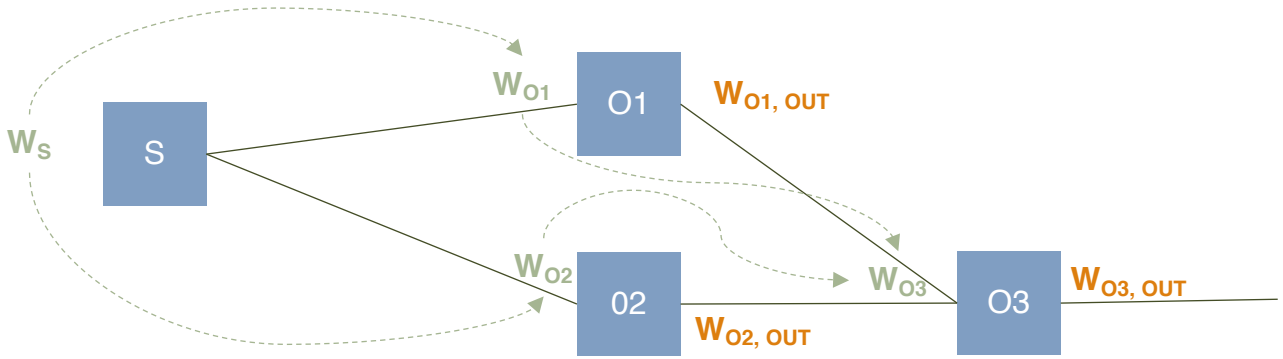
We call the value of this monotonic function F^* the (low) **watermark** of operator O!

✓ Monotonicity ➔ no tuples with $ts < W$ will arrive in the future.

✓ Solves problem of safely closing windows!

* The watermark of operator O is function of O and all its upstream peers, but we omit the latter for brevity.

Watermarks in Practice



Input Watermark: Earliest ts that O can receive.

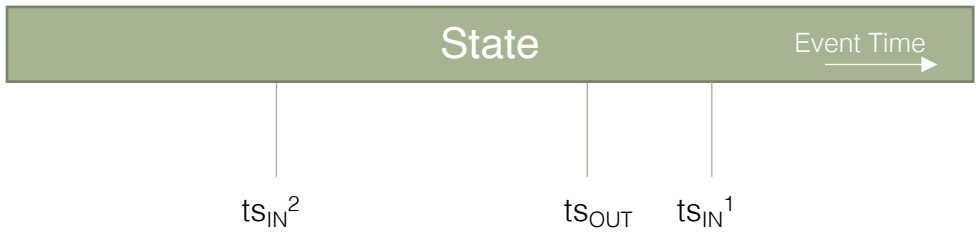
Output Watermark: Earliest ts that O can emit.

✓ Watermarks are **generated** at the sources.

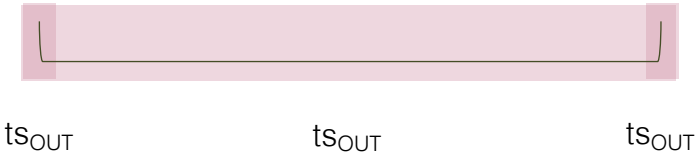
✓ They (conceptually) **flow** through the pipeline.

✓ They propagate **regardless of data filtering** ➔ all operators have up-to-date view of time

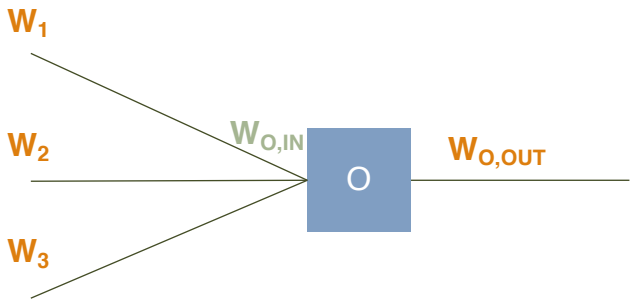
Input & Output Timestamps



1. Which windows are complete?
2. How is the timestamp set for window results?

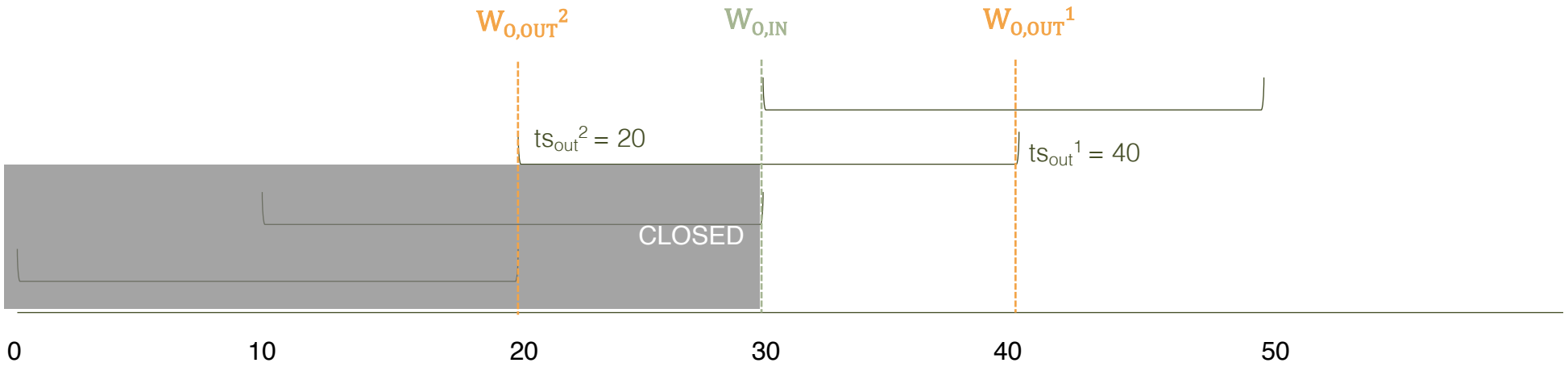


Computing Watermarks

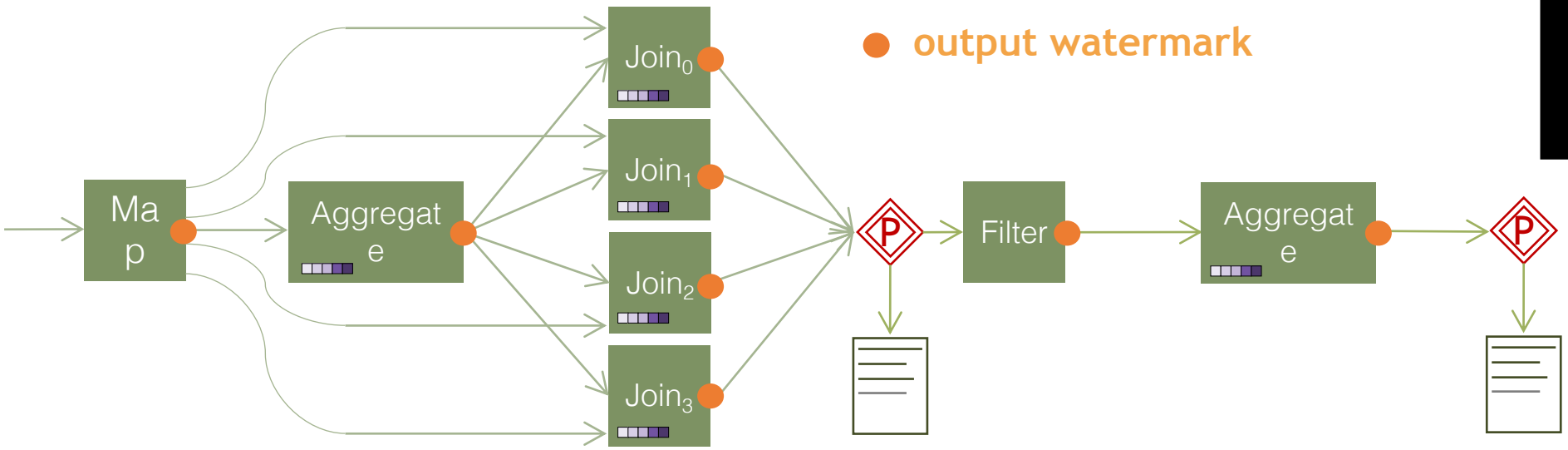


$$W_{O,IN} = \min(W_1, W_2, W_3)$$

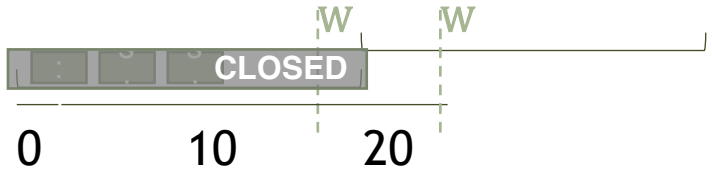
$$W_{O,OUT} = \begin{cases} W_{O,IN} \text{ if O stateless} \\ g(state_O, semantics_O) \text{ otherwise} \end{cases}$$



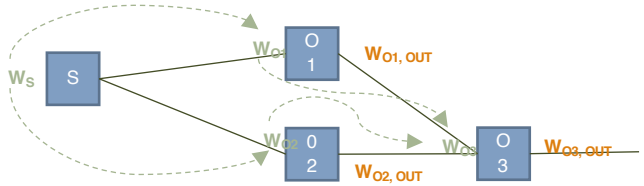
Flink Example



1. Result Correctness from correct window assignment



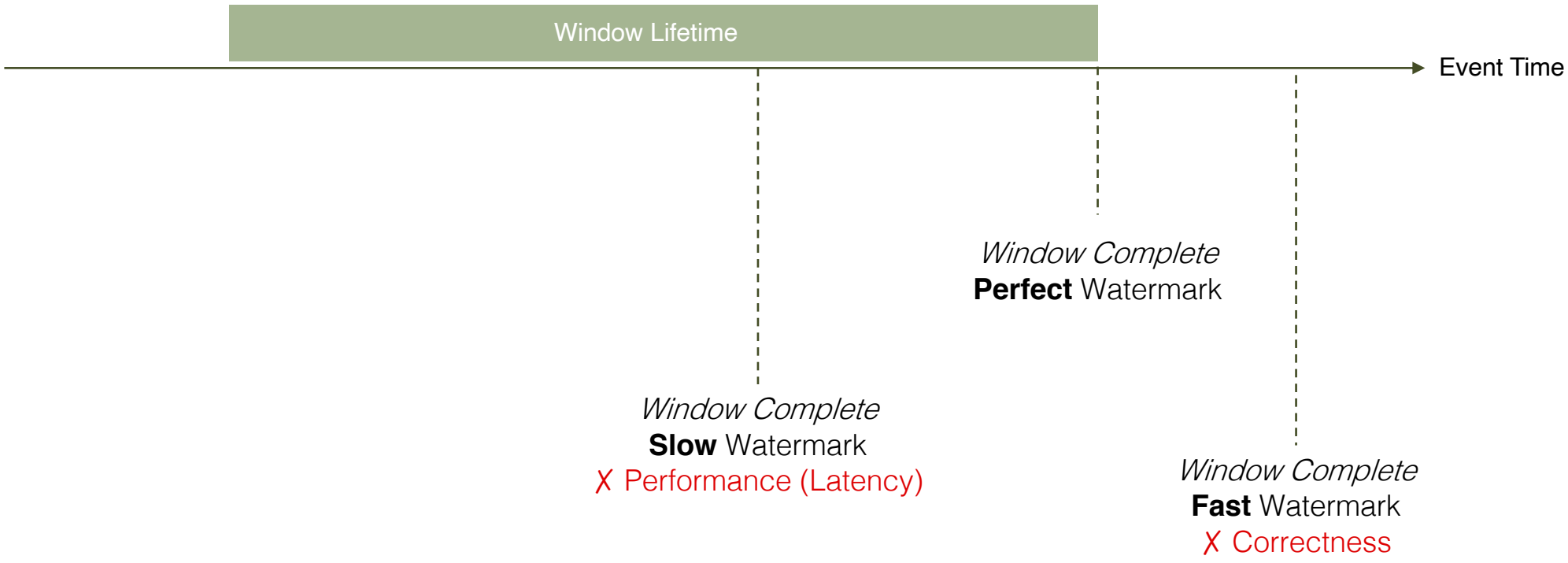
2. Watermark propagation



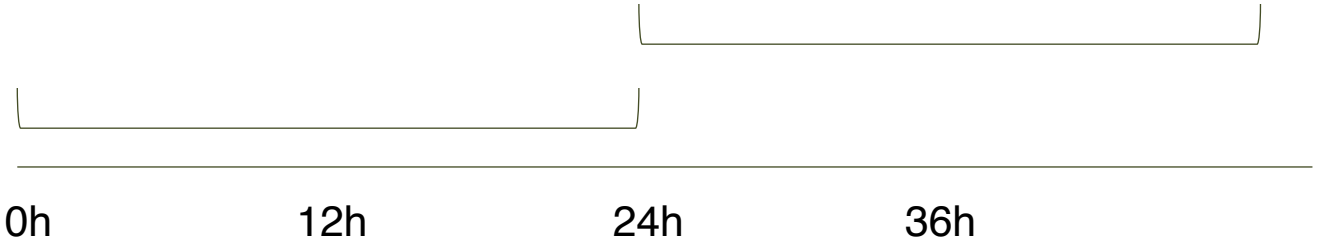
Generating Watermarks

- **Perfect Watermarks**
  - Sorted data or very predictable data sources.
  - Determinism without sorting for order-independent window functions.
  - Disorder inside windows is still possible!
  - Sorting possible if needed, but not imposed.
- **Heuristic Watermarks**
  - When impossible to perfectly predict data (e.g., distributed sources).
  - Best-effort prediction of event-time progress.
  - Possibility for late data.
  - More knowledge about internals of sources → less mispredictions (late data).

Fast and Slow Watermarks



Triggering



Latency of 24 hours!
Can we do better?

Emit (partial) result every 1h

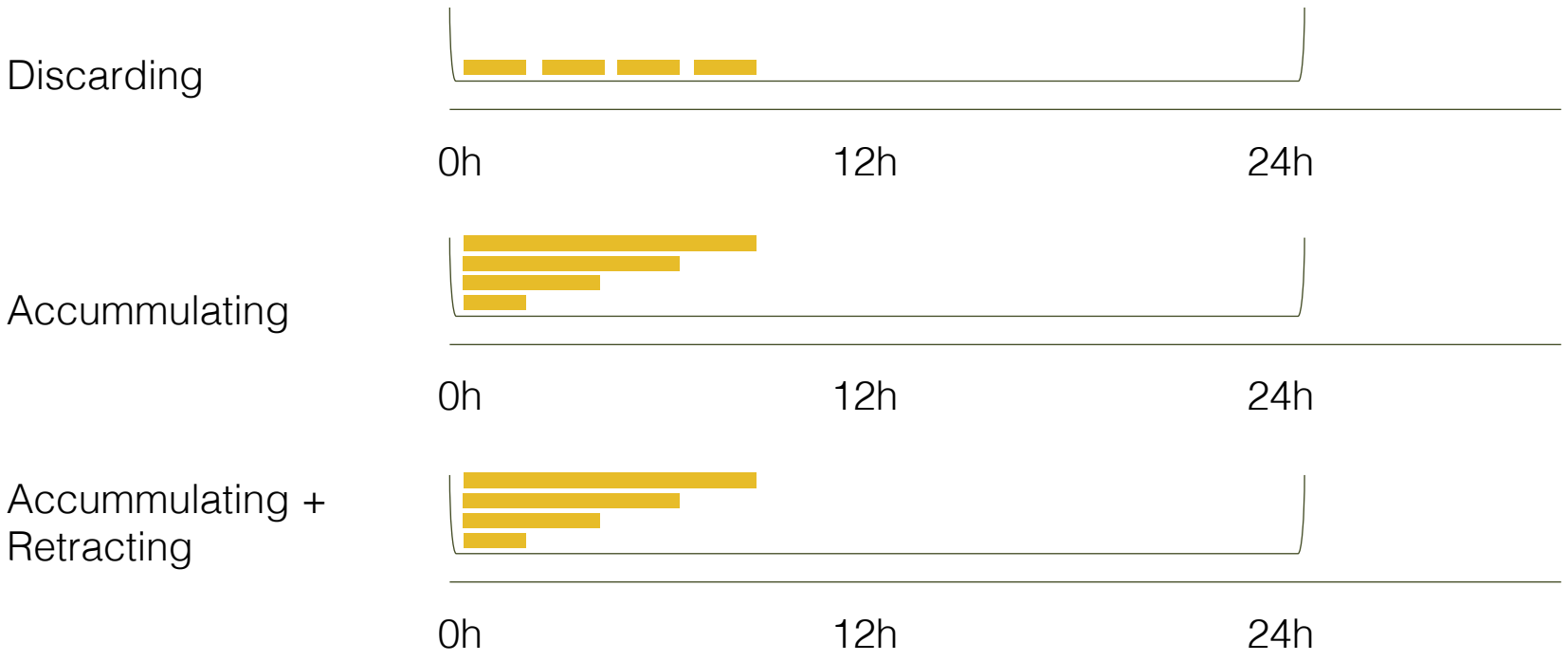
Correctness — **Performance Flexibility**

Completeness Trigger
Emit result when window is complete

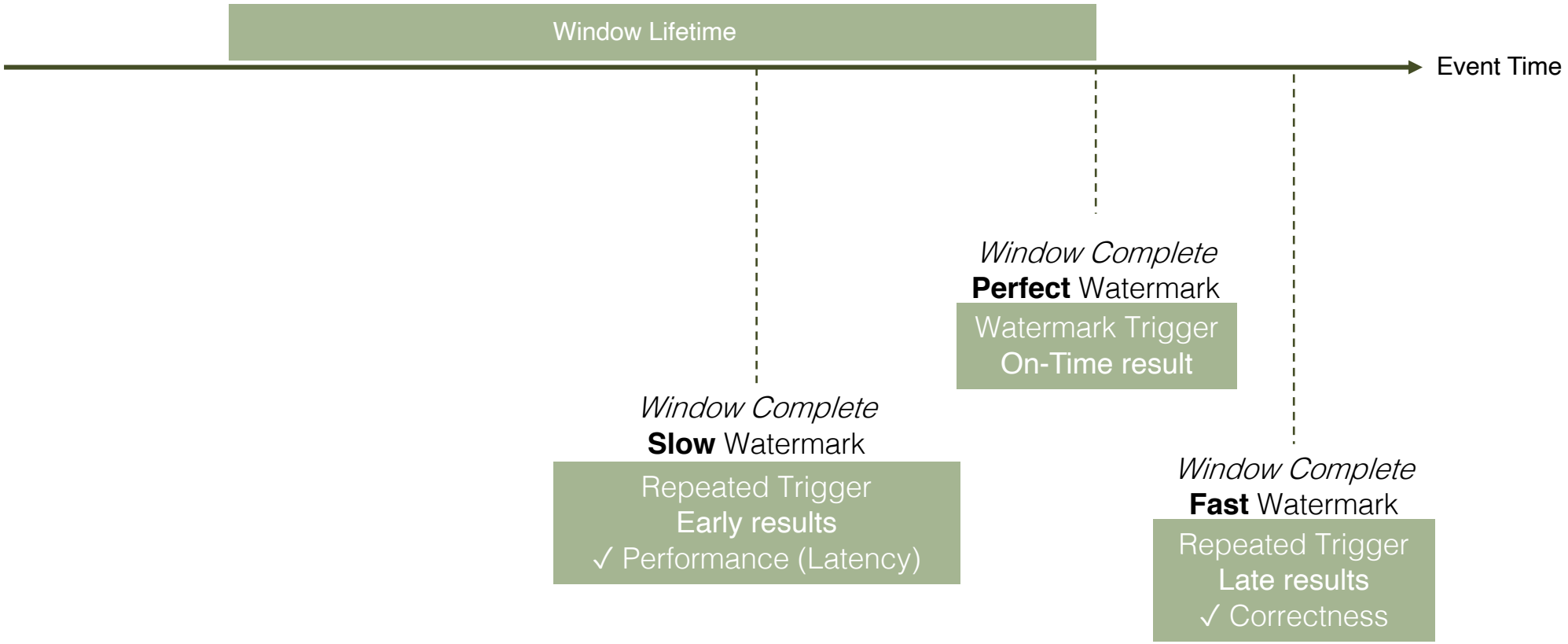
Repeated Update Trigger
Periodically emit (partial) result
(e.g., for every tuple, every hour, etc)

Repeated Update Trigger Results

Repeated Update Trigger ➔ Every 1h



Putting It All Together



… the light at the end of the tunnel …

- Motivation, preliminaries and examples about data streaming and Stream Processing Engines
- Causes of out-of-order data and solutions enforcing total-ordering
- Pros/Cons of total-ordering
- Relaxation of total-ordering and the watermarks

To summarize:

Event time advances based on:	Cons	Pros
Tuples themselves	• Costly merge-sorting • Coupled processing / output of tuples	• Determinism, for order sensitive / insensitive functions
Watermarks	• Would require special support for order-sensitive functions • Latency depends on frequency of watermarks	• Decoupled processing / output of tuples • Propagation of time passing even in the absence of tuples

Tutorial:

The Role of Event-Time Order in Data Streaming Analysis

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Bastian Havers
Chalmers University of Technology & Volvo Cars
Gothenburg, Sweden
havers@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se