

Vertically Scalable solutions illustrated via EPL and Esper

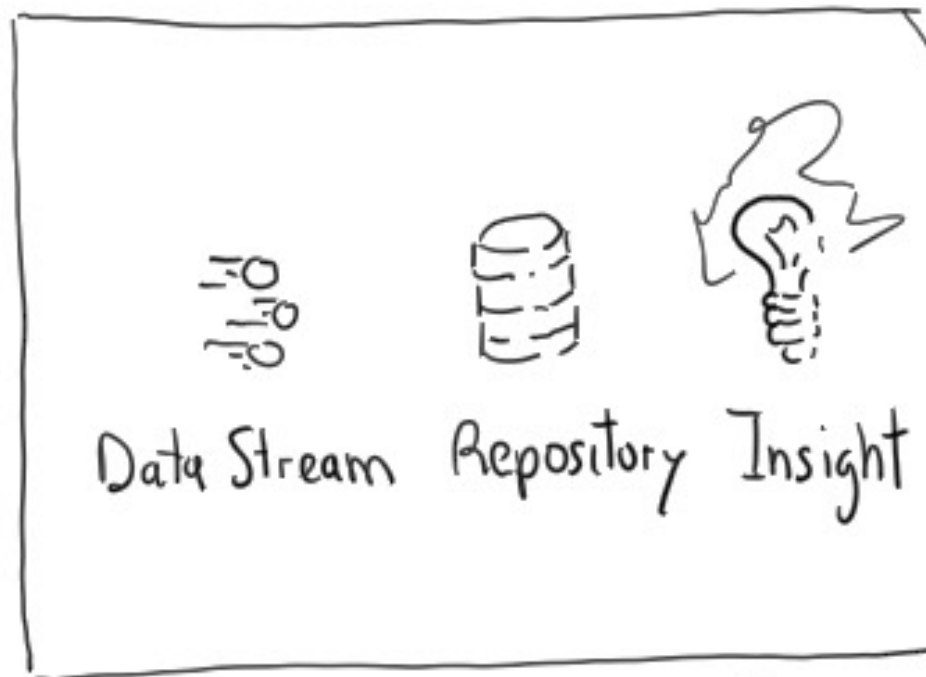
Emanuele Della Valle

Politecnico di Milano

Positioning this lecture

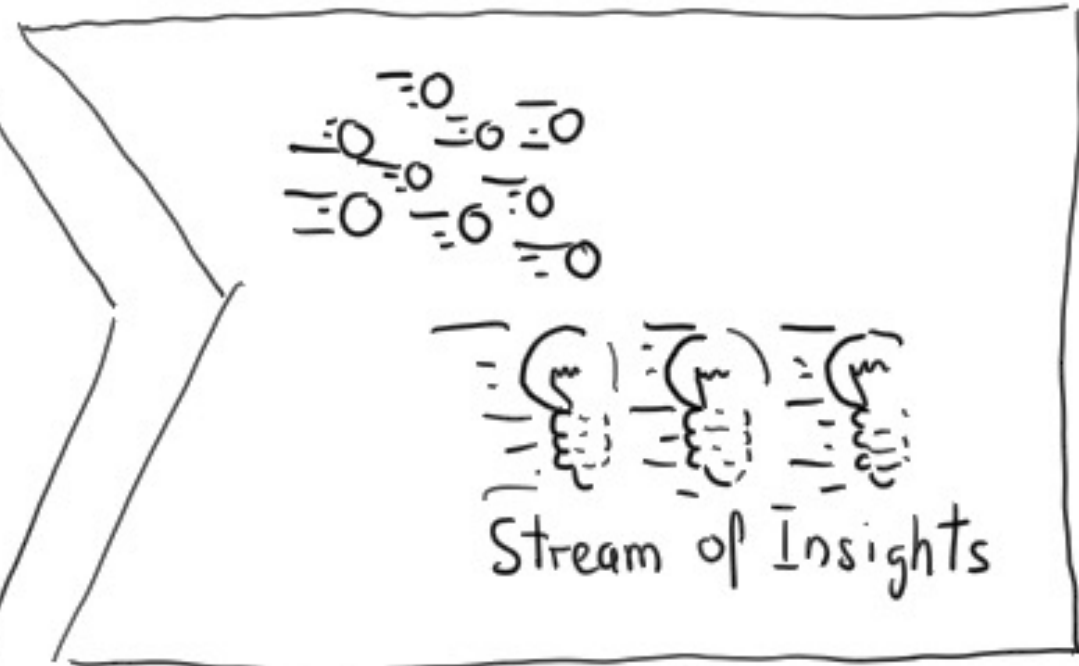
1st premises: continuity matters

Traditional approach

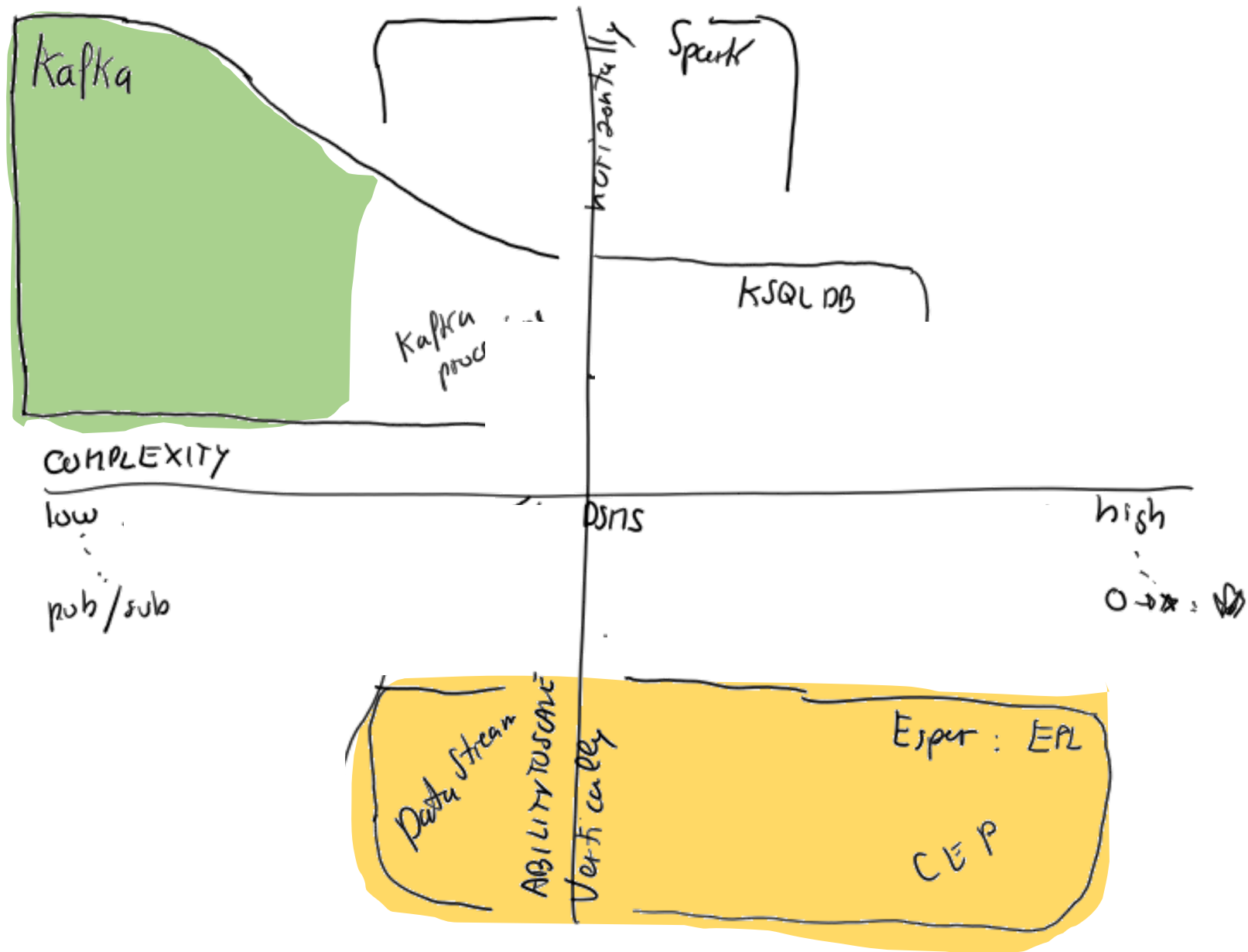


Stop data to analyse

Velocity approach



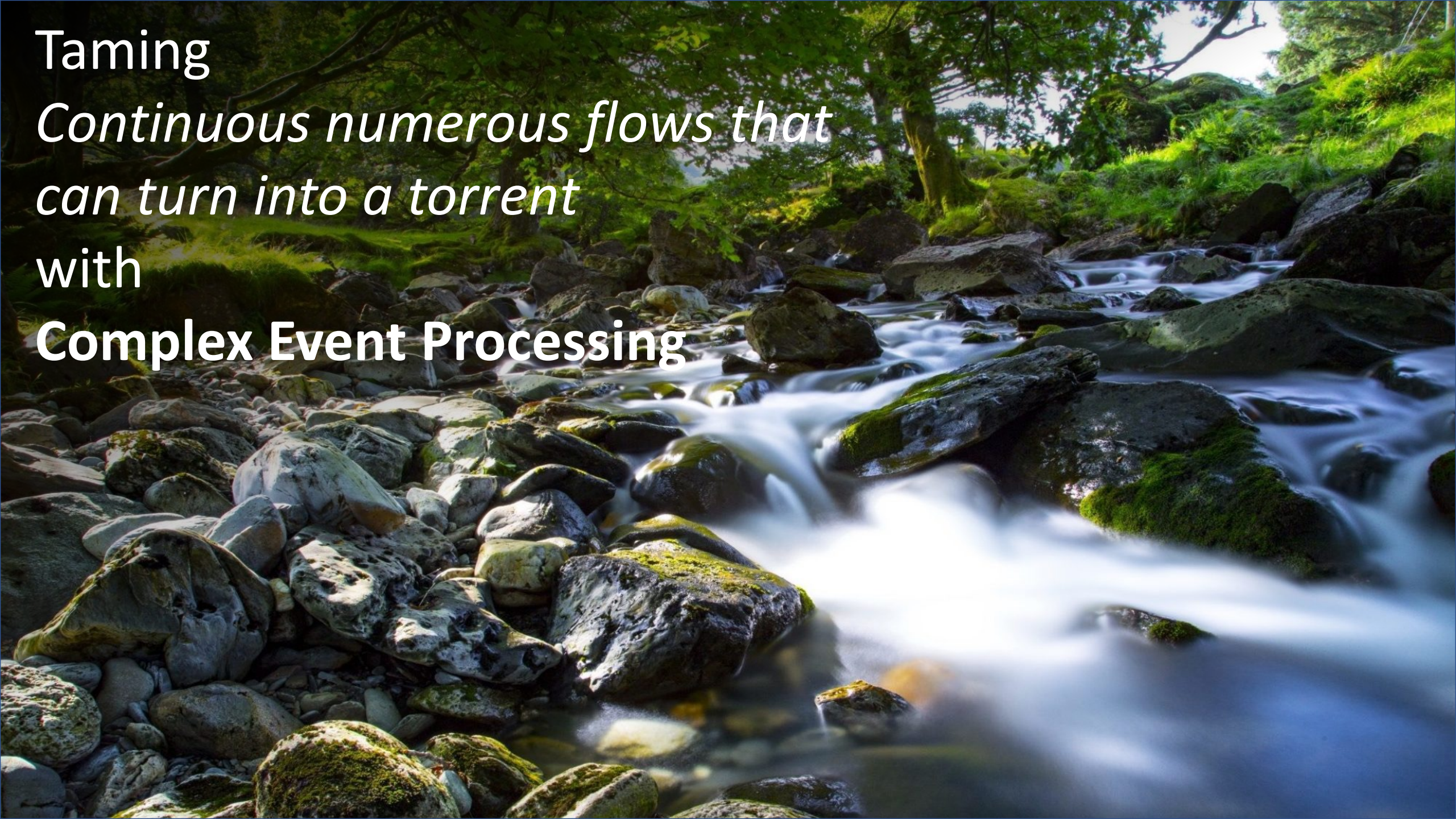
Analyse data in motion



Taming

*Continuous numerous flows that
can turn into a torrent
with*

Complex Event Processing



The Event Processing Language & its reference implementation Esper

Esper in a nutshell

- Implemented as a Java library
 - Can be embedded in any JVM application
- Designed for performance
 - High throughput
 - Low latency
- Tree-based recognition algorithm
- Inverted indexes to dispatch incoming events to EPL statements
- Builds indexes to quickly retrieve events with given properties among those stored

Esper in a nutshell (cont.)

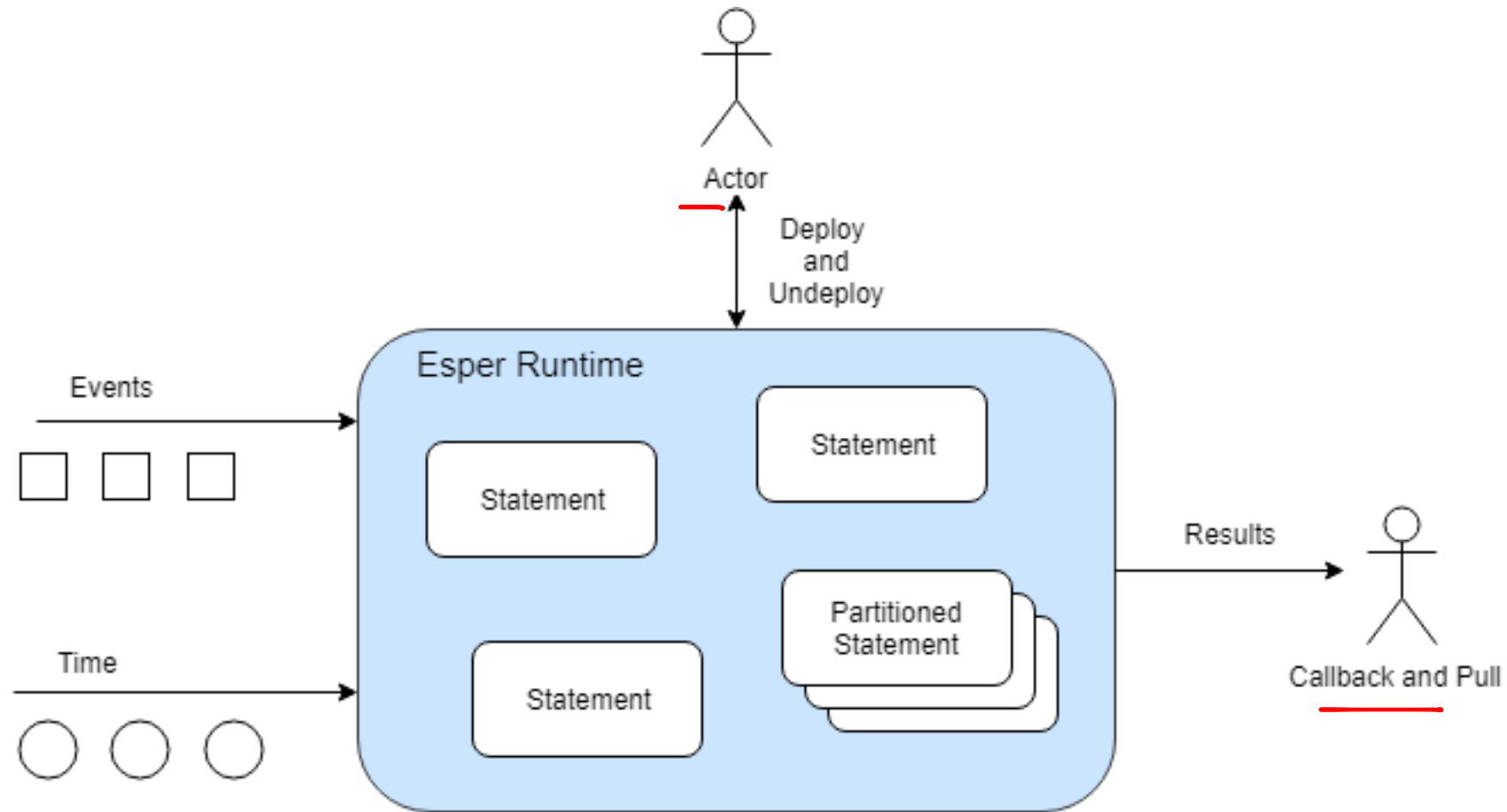
- Interaction with static / historical data
- Configurable push or pull communication
- Several adapters for input/output
 - CSV, JMS in/out, API, DB, Socket, HTTP
- Esper HA
 - High Availability
 - Ensures that the state is recoverable in the case of failure

€

EPL in a nutshell

- EPL: rich language to express rules (a.k.a., statements)
- Grounded on the DSMS approach
 - Windowing
 - Relational select, join, aggregate, ...
 - Relation-to-stream operators to produce output
- Queries can be combined to form a graph
- Includes complex event recognition abstractions
 - Pattern detection

Esper & EPL



The Basics of the Event Processing Language

Running example

- Count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes
- Events
 - Smoke event: String sensor, boolean state
 - Temperature event: String sensor, double temperature
 - Fire event: String sensor, boolean smoke, double temperature
- Condition:
 - Fire: at the same sensor smoke followed by temperature>50 within 2 minutes

Declare event types

- Two ways
 - EPL *create schema* clause
 - Runtime configuration API *addEventType*

- Syntax

```
create schema  
schema_name [as]  
(property_name property_type  
[, property_name property_type [, ...]])  
[inherits inherited_event_type  
[, inherited_event_type] [, ...]]
```


Running example

```
create schema SmokeSensorEvent(  
    sensor string,  
    smoke boolean  
);
```

```
create schema TemperatureSensorEvent(  
    sensor string,  
    temperature double  
);
```

```
create schema FireComplexEvent(  
    sensor string,  
    smoke boolean,  
    temperature double  
);
```

Event Processing Language (EPL)

- EPL is similar to SQL *DSMS*
 - Select, where, ...
- Event streams and views instead of tables
 - Views define the data available for the query
 - Views can represent windows over streams
 - Views can also sort events, derive statistics from event attributes, group events, ...

EPL syntax

```
[insert into insert_into_def
select select_list
from stream_def [as name]
[, stream_def [as name]] [, ...]
```

0
win

```
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

0 →

| ← different sem.

0 EPL

Running example

<http://esper-epl-tryout.appspot.com/epltryout/mainform.html>

EPL Statements

EPL Module Text
Enter EPL Here:

```
create schema SmokeSensorEvent(sensor string, smoke boolean);

create schema TemperatureSensorEvent(sensor string, temperature double);

create schema FireComplexEvent(sensor string, smoke boolean, temperature double);

insert into FireComplexEvent
select a.sensor as sensor, a.smoke as smoke, b.temperature as temperature
from pattern [every a=SmokeSensorEvent(smoke=true) ->
b=TemperatureSensorEvent(sensor=a.sensor, temperature>50)];

select count(*)
from FireComplexEvent.win:time(10 min);
```

Time And Event Sequence

Beginning Of Time
Provide a timestamp to start at:

2001-01-01 08:00:00.000 **Submit**

Advance Time and Send Events
Enter sequence of time and events:

```
SmokeSensorEvent={sensor='S1', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=30}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S1', smoke=true}
TemperatureSensorEvent={sensor='S1', temperature=40}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S2', smoke=false}
TemperatureSensorEvent={sensor='S1', temperature=55}

t=t.plus(11 min)
```

Scenario Results

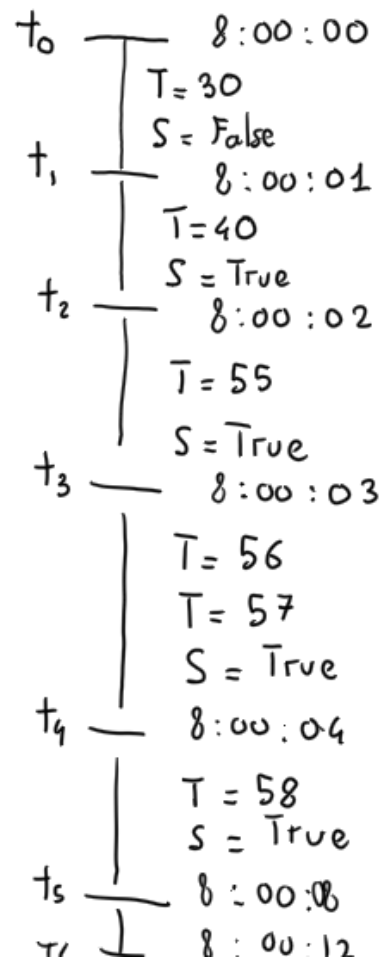
All Output Events
Output Per Statement **All Audit Text**
Audit Text Per Statement

At: 2001-01-01 08:00:02.000
Statement: Stmt-4
Insert
FireComplexEvent={sensor='S1', smoke=true, temperature=55.0}

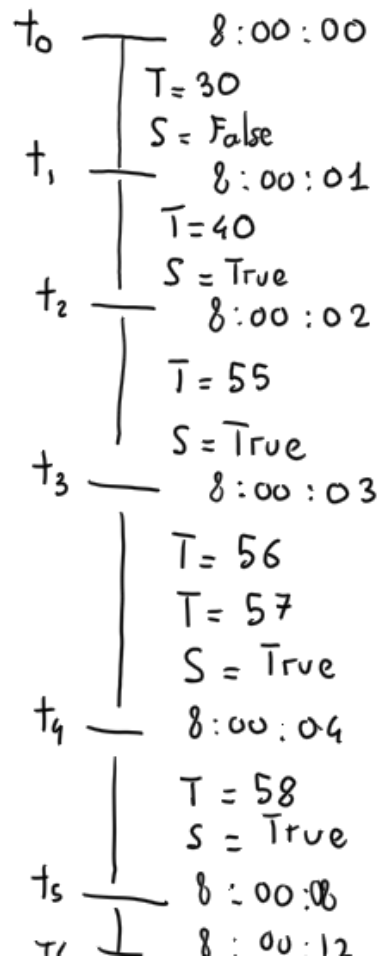
Statement: Stmt-5
Insert
Stmt-5-output={count(*)=1}

At: 2001-01-01 08:10:02.000
Statement: Stmt-5
Insert
Stmt-5-output={count(*)=0}

Time line



Filtering



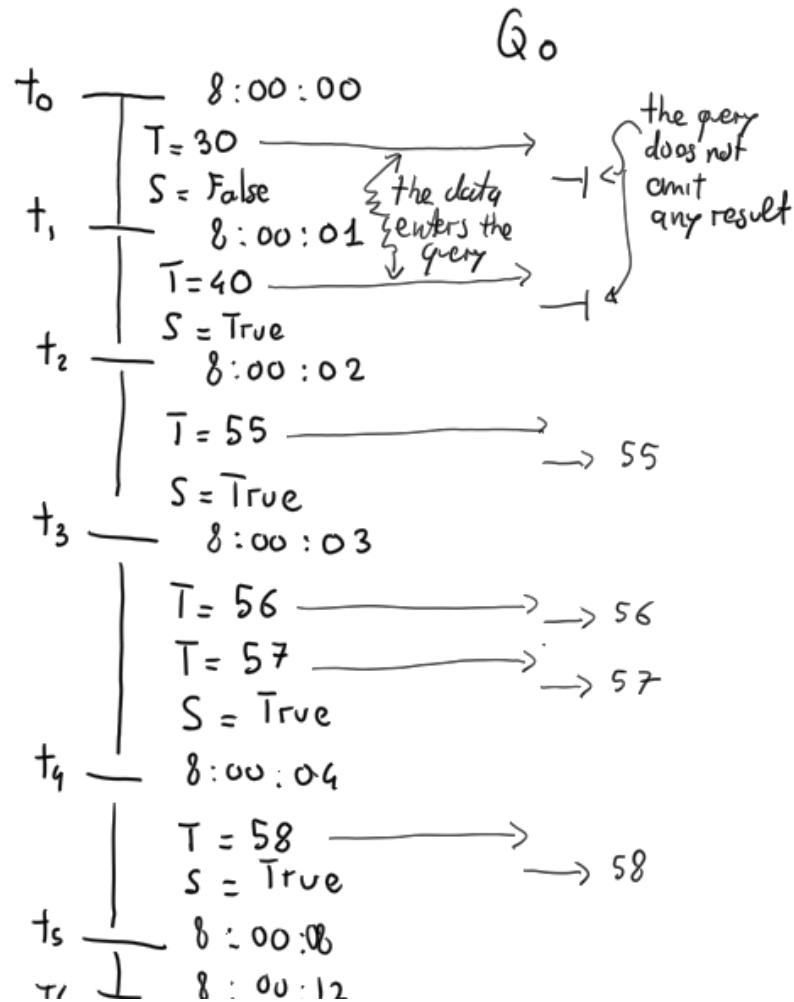
Q0 - the SQL-style

```
select *  
from TemperatureSensorEvent  
where temperature > 50;
```

Q0bis - the Event-Based System Style

```
select *  
from TemperatureSensorEvent (temperature > 50) ;
```

Filtering the SQL style



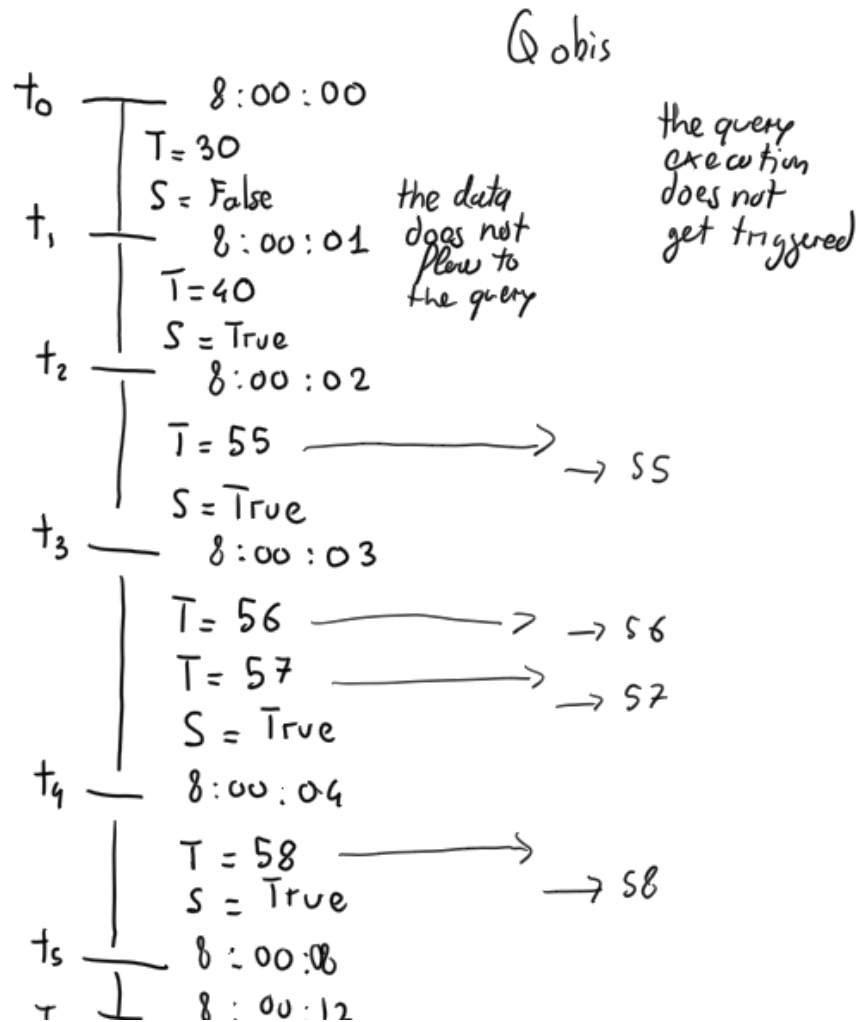
Q0 - the SQL-style

```
select *  
from TemperatureSensorEvent  
where temperature > 50;
```

Execution mode

Data points enter the query that filters them

Filtering the Event-Based System Style



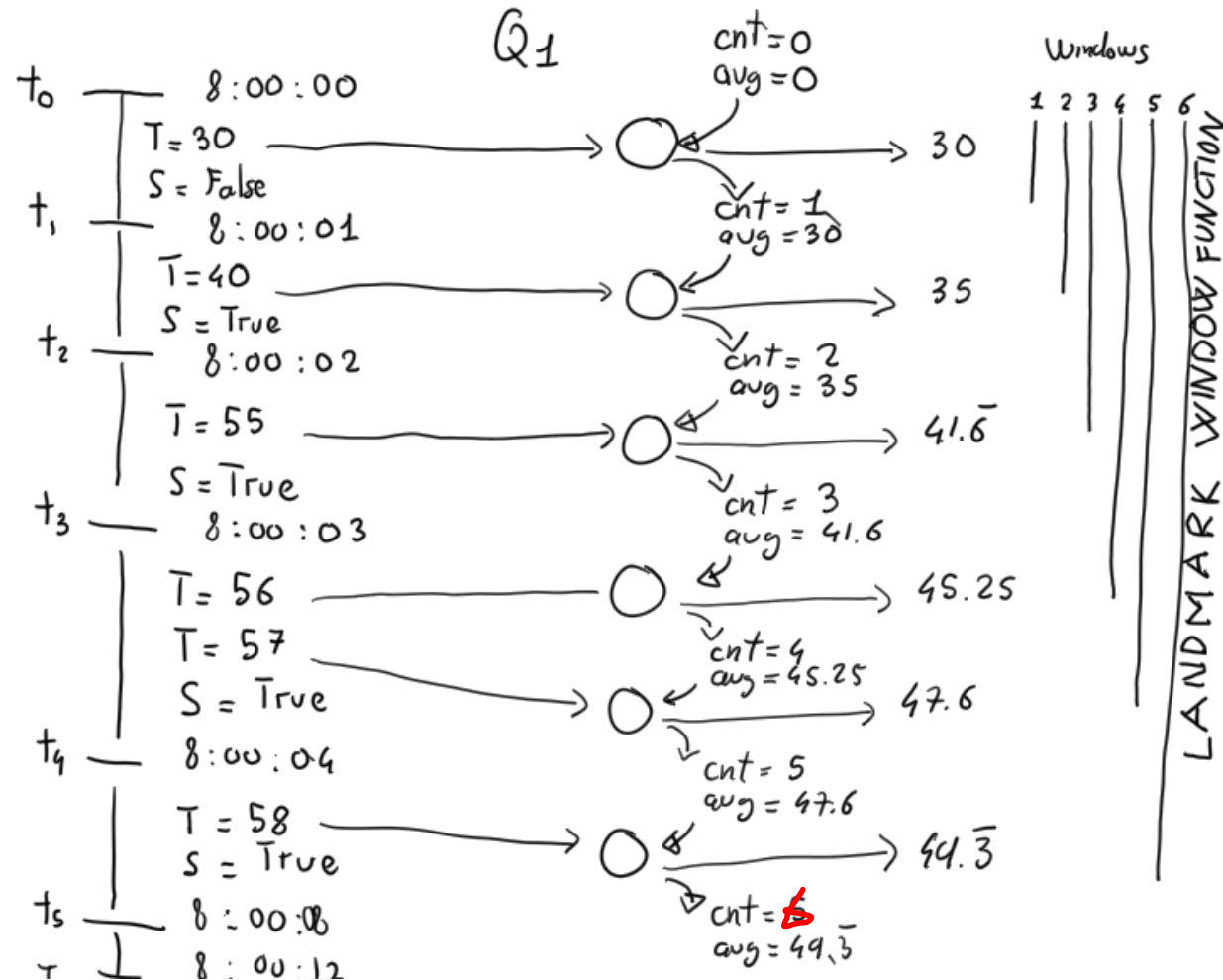
Q0bis - the Event-Based System Style

```
select *  
from TemperatureSensorEvent(temperature > 50)  
;
```

Execution mode

Data points are filtered before flowing into the query and the query execution does not get triggered

Basic Aggregation



```
select sensor, avg(temperature)
from TemperatureSensorEvent
group by sensor;
```

Execution mode

From a **logical** perspective, **data points cumulate in the landmark window**, and the query emits a new avg for every data point

From a **physical** perspective, **the query is evaluated incrementally** and **maintains a state** that captures the number of events (cnt) seen so far and the previous avg. The new average only depends on the state and the new data point entering the query. Old data can be forgotten.

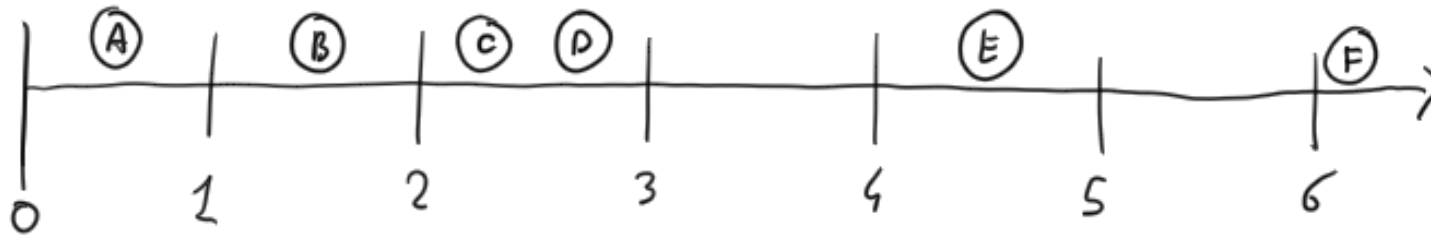
Data Windowing

	Physical # events	Logical # time units
sliding moves on <u>every event</u>	native ✓	native ✓
tumbling	native ✓	native ✓
hopping	output clause ✓	output clause ✓

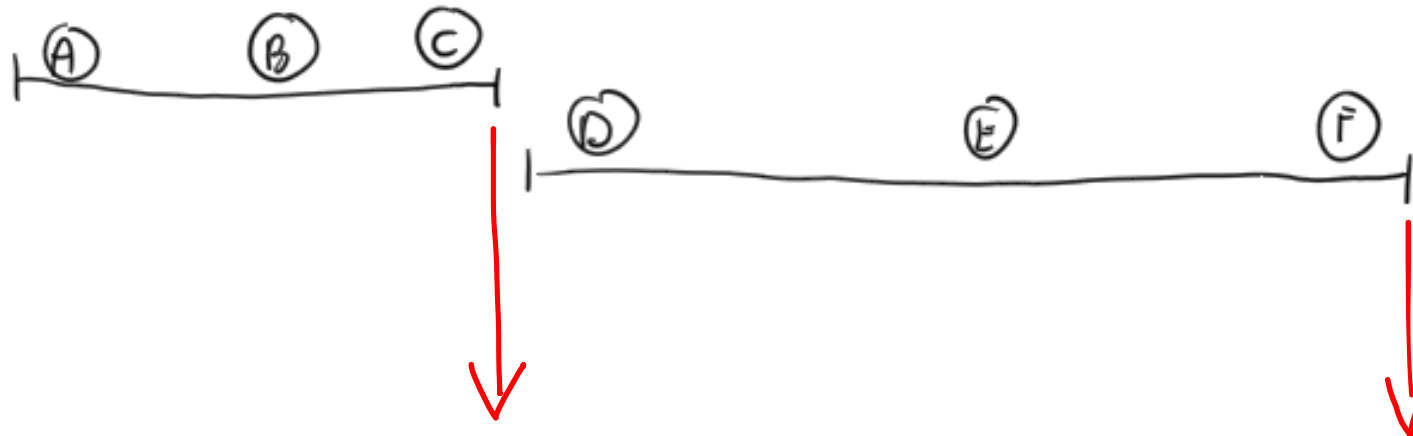
+ session window

Physical tumbling windows

Q5

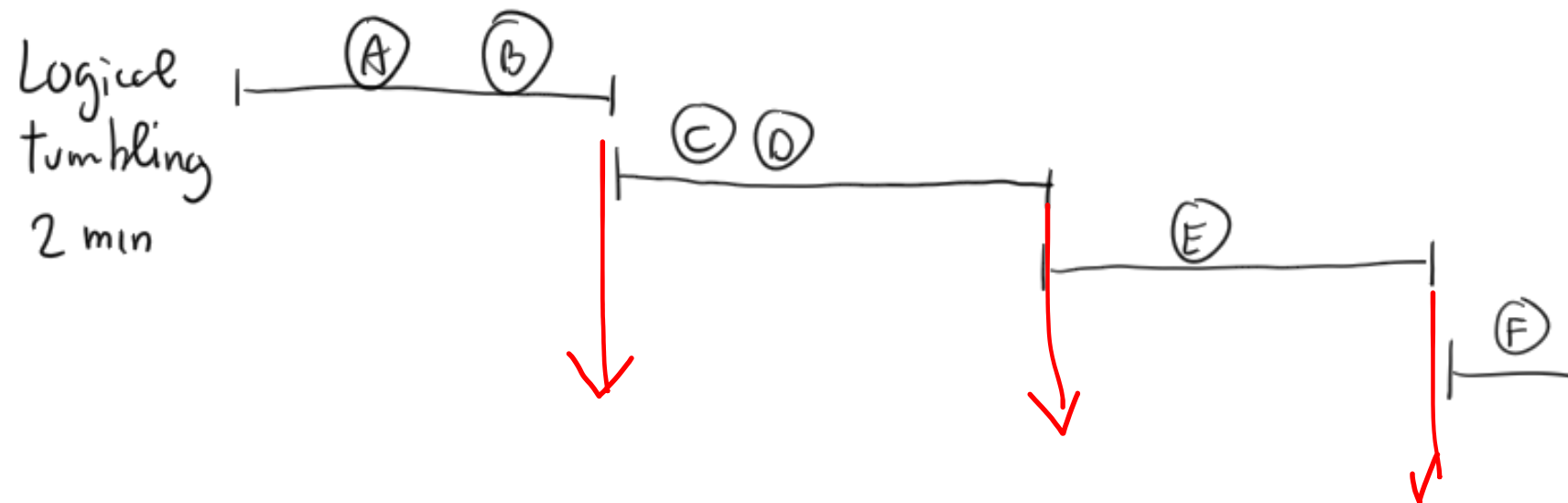


Physical
tumbling
3 events



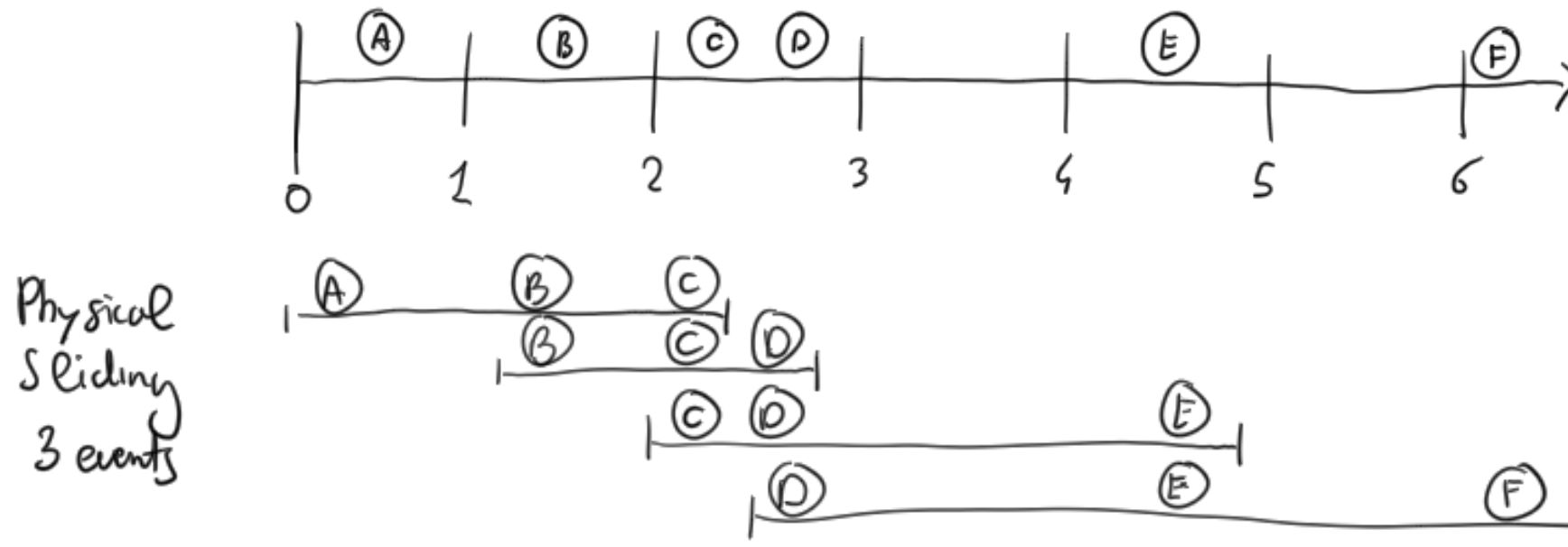
Logical tumbling windows

Q₃



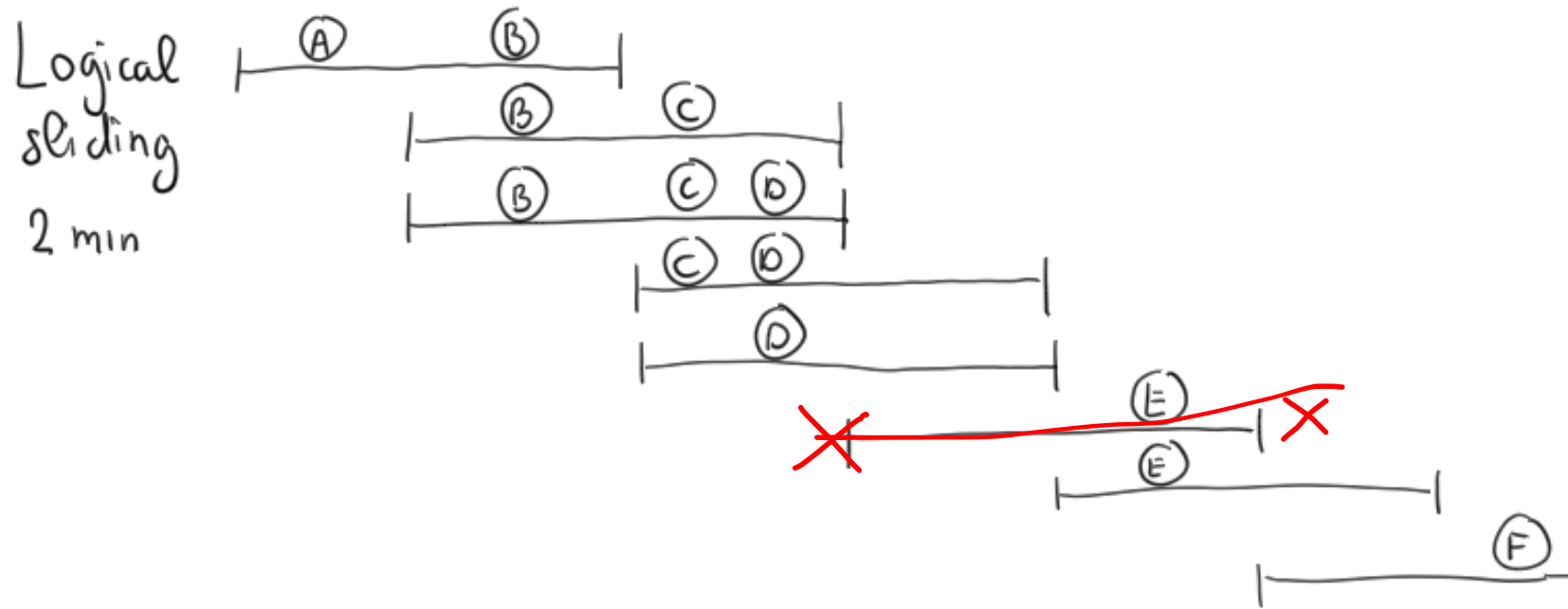
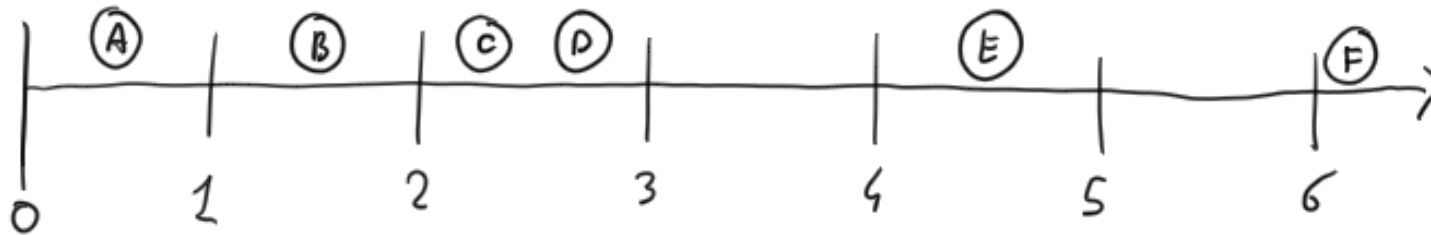
Physical sliding windows

Q₄



Logical sliding windows

Q₂

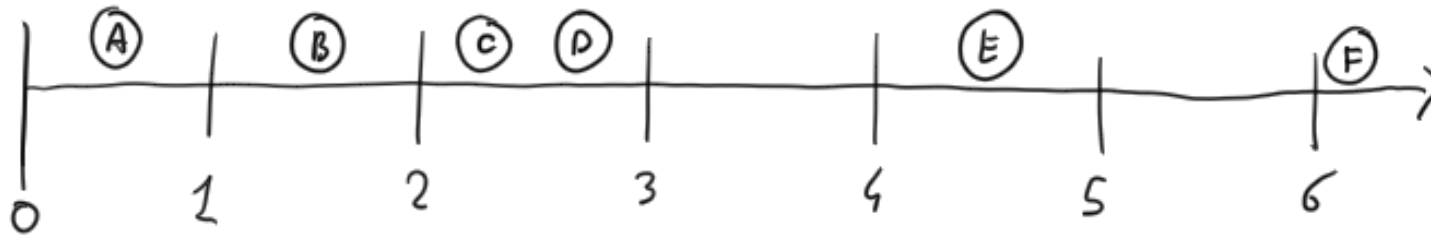


EPL window syntax

Type	Syntax	Description
Logical Sliding	<code>win:time(<u>time_period</u>)</code>	Sliding window that covers the specified time interval into the past
Logical Tumbling	<code>win:time_batch(<u>time_period</u> [<u>reference point</u>] [<u>flow control</u>])</code>	Tumbling window that <u>batches events</u> and releases them every specified time interval, with flow control options
Physical Sliding	<code>win:<u>length</u>(size)</code>	Sliding window that covers the specified number of elements into the past
Physical Tumbling	<code>win:length_batch(<u>size</u>)</code>	Tumbling window that batches events and releases them when a given minimum number of events has been collected

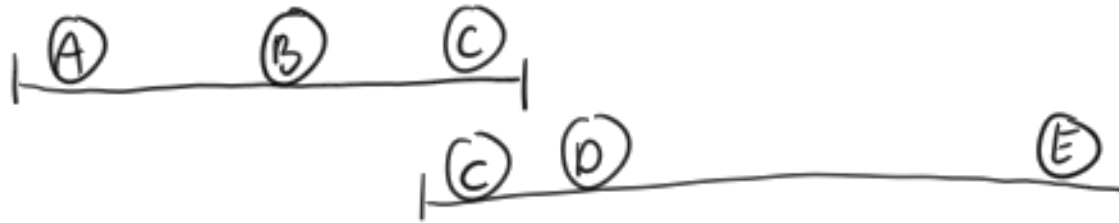
Physical hopping windows

16:25



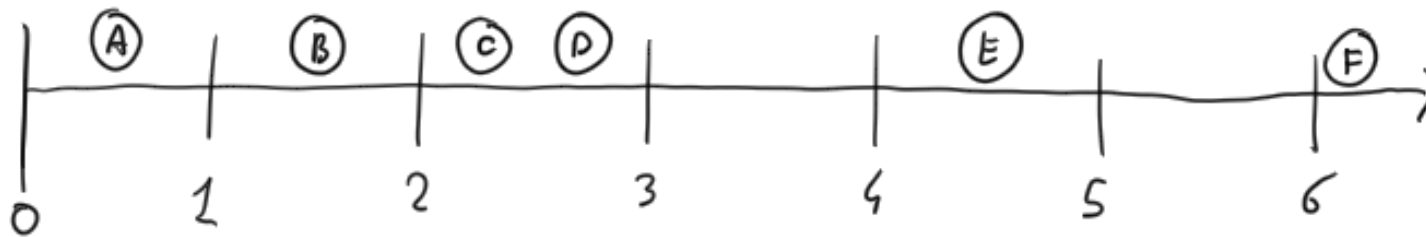
Q_{6b15}

Physical
hopping
3 events
updated
every
2 events

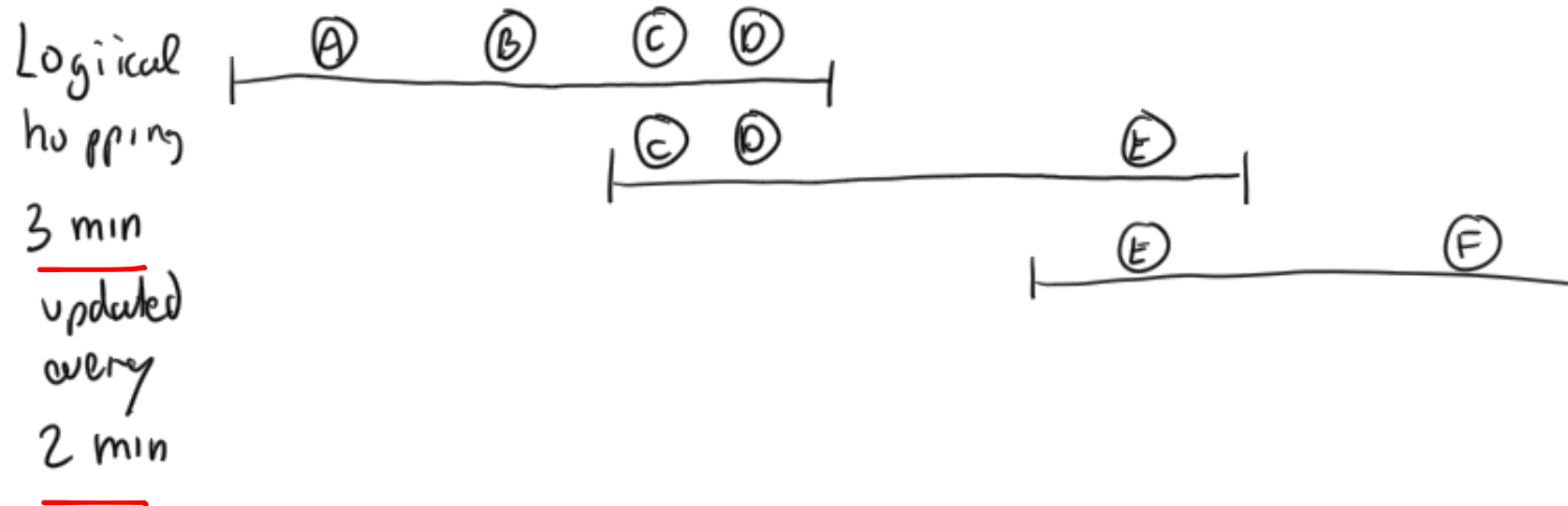


Logical hopping windows

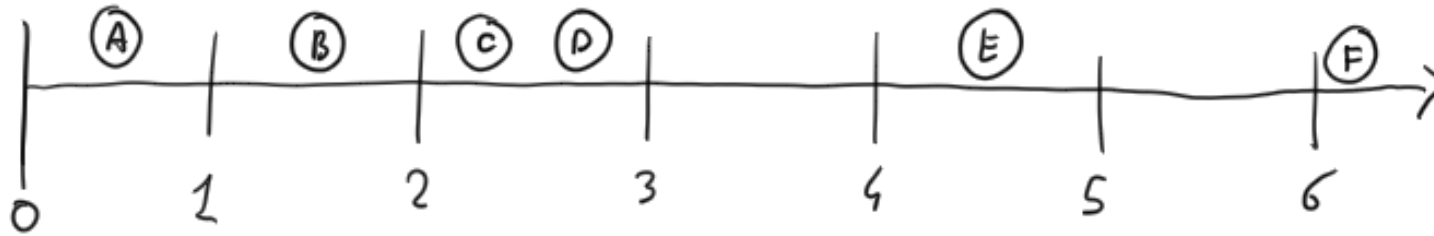
Q₆



7



Session windows (not supported in EPL)



Session
Window
less than
1 min
apart



EPL Output control syntax

~~10.50~~

- The *output* clause is optional in Esper
- It is used to
 - Control the output rate
 - Suppress output events

```
output      [all | first | last | snapshot]  
everyoutput_rate [seconds | events]
```

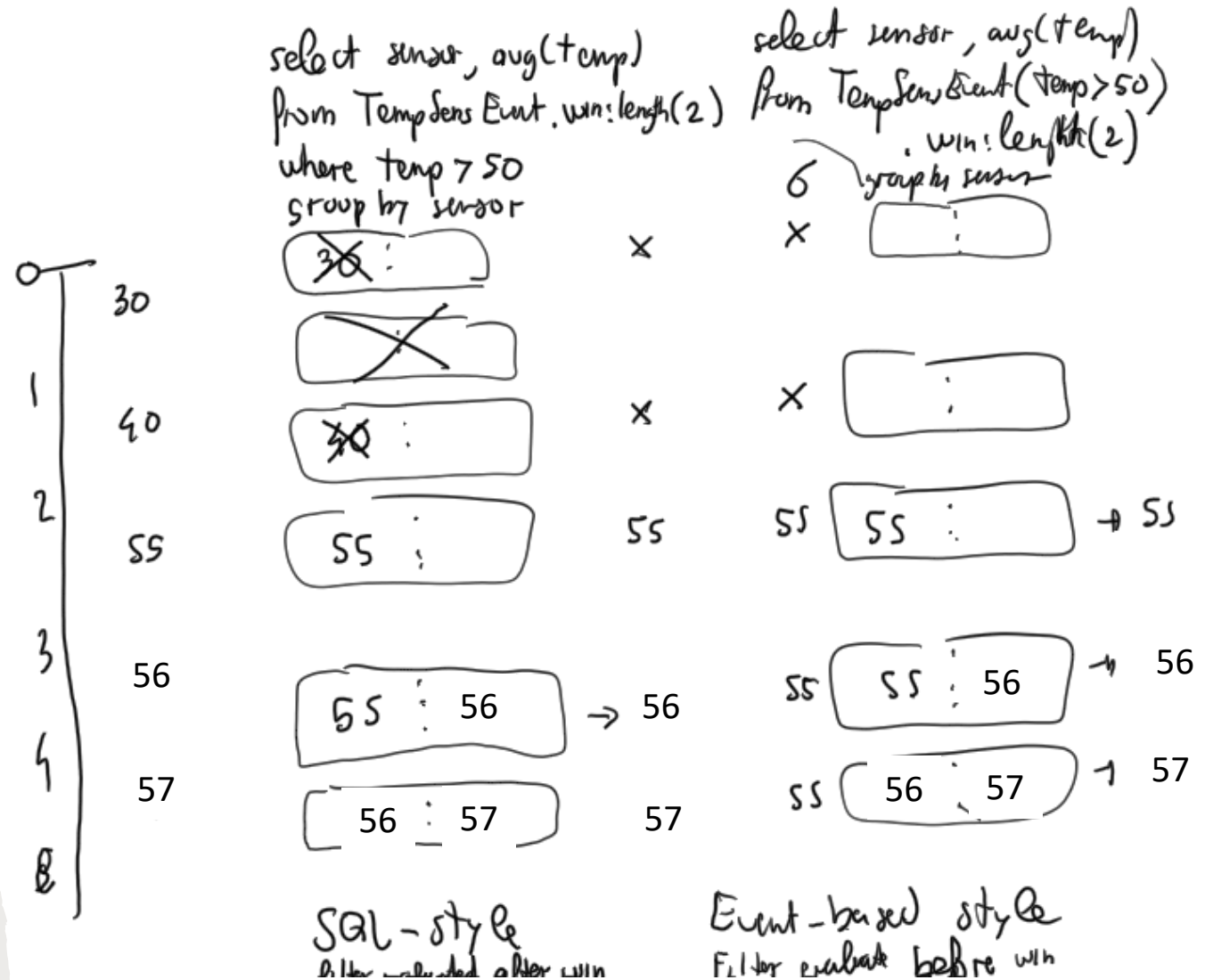
Output control and Hopping Windows

- Control advancement of sliding windows

```
select      avg(temperature)
from        TemperatureSensorEvent.win:length(4)
output      snapshot every 2 events
```

```
select      avg(temperature)
from        TemperatureSensorEvent.win:time(4 sec)
output      snapshot every 2 sec
```

A digression on data windowing and SQL-style vs. Event- base style filtering



The Pattern Matching clause of the Event Processing Language

Pattern matching

An event pattern emits when one or more event occurrences match the pattern definition, which can include

- Constraints on the content of events
- Constraints on the time of occurrence of events
- Conditions for pattern creation / termination


Pattern matching

- Content-based event selection

```
TempStream(sensor="S0", val>50)
```

- Time-based event observers specify time intervals or time schedules

```
timer:interval(10 seconds)
```



Fires after 10 seconds

```
timer:at(5, *, *, *, *)
```



Every 5 minutes

Syntax: minutes, hours, days of month, months, days of week

Pattern matching operators

- Logical operators
 - and, or, not
- Temporal operators that operate on event order
 - -> followed-by
- Creation/termination control
 - every, every-distinct, [num] and until
- Guards filter out events and cause termination
 - timer:within, timer:withinmax and while-expression

Pattern matching

```
select a.sensor from pattern
[every (
  a = SmokeEvent(smoke=true)
  ->
  TempEvent(val>50, sensor=a.sensor)
  where timer:within(2 min)
)]
```

Pattern matching

17:30

- every *expr*
 - When *expr* evaluates to true or false ...
 - ... the pattern matching for *expr* should re-start
- Without the every operator the pattern matching process does not re-start

Pattern matching

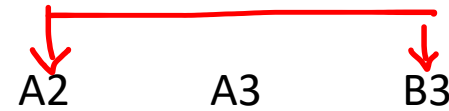
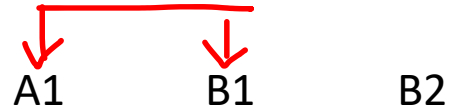
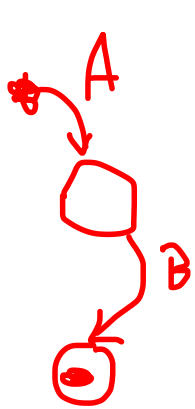
- This pattern fires when encountering an A event and then stops

A

- This pattern keeps firing when encountering A events, and does not stop

every A

Pattern matching

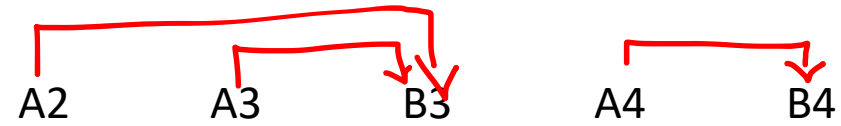
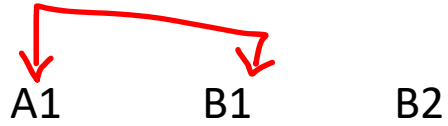
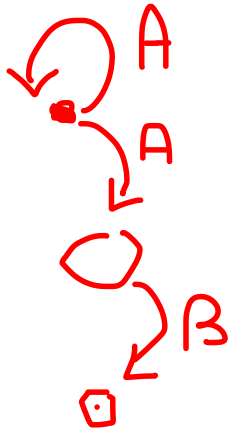


every (A \rightarrow B)

Detect an event A followed by an event B:
at the time when B occurs, the pattern
matches and restarts looking for the next
A event

B1	{A1, B1}
B3	{A2, B3}
B4	{A4, B4}

Pattern matching

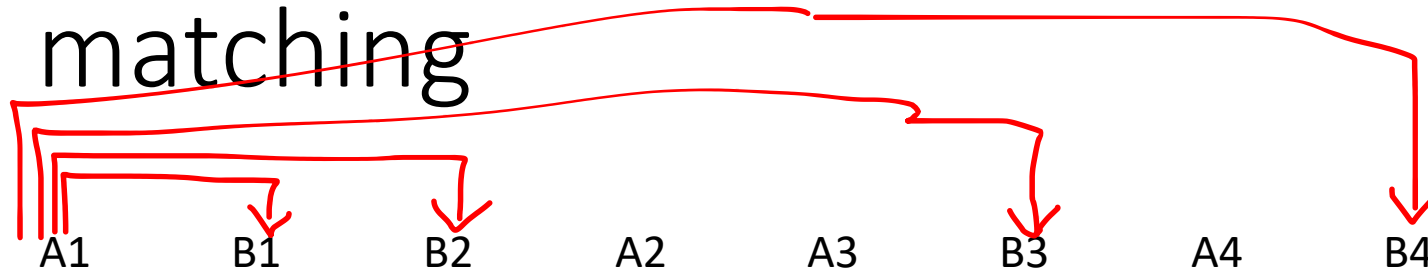
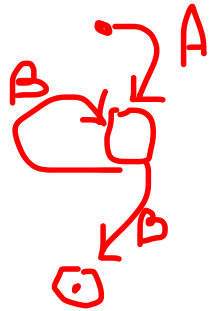


every $A \rightarrow B$

The pattern fires for every A followed by a B event

B1	{A1, B1}
B3	{A2, B3}, {A3, B3}
B4	{A4, B4}

Pattern matching

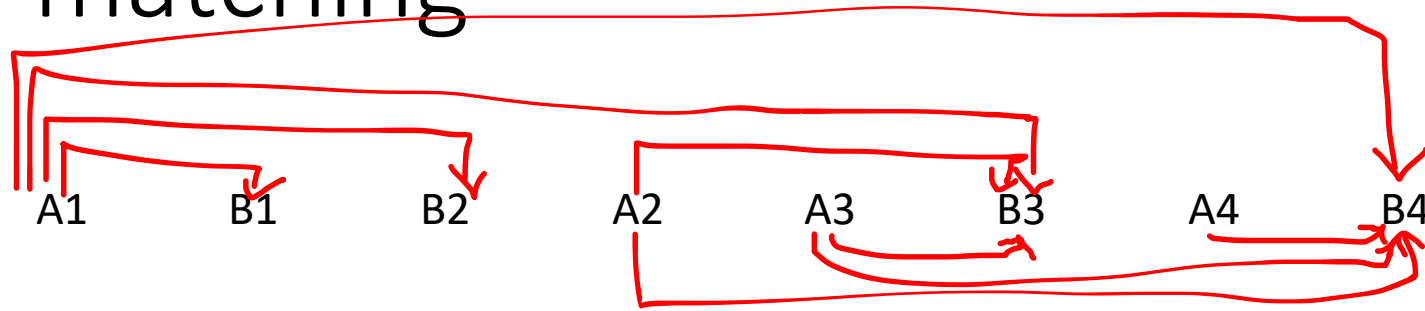
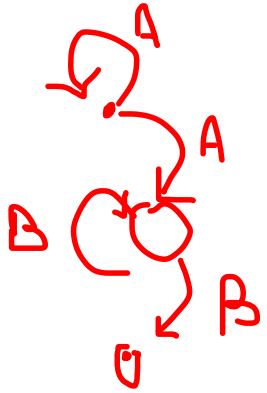


A -> every B

The pattern fires for an A event followed by every B event

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}
B4	{A1, B4}

Pattern matching



every A -> every B

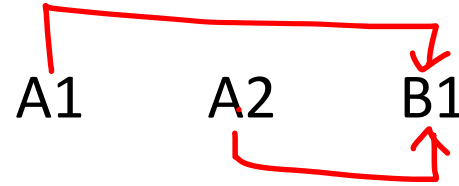
The pattern fires for every A event followed by every B event

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}, {A2, B3}, {A3, B3}
B4	{A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}

Pattern matching

- With the every operator
 - Multiple (partial) instances of the same pattern can be active at the same time
 - Each instance can consume some resources when events enter the engine
- End pending instances whenever possible
 - With the *timer:within* construct
 - With the *and not* construct
- Note: the data windows on a pattern do not always limit pattern sub-expression lifetime

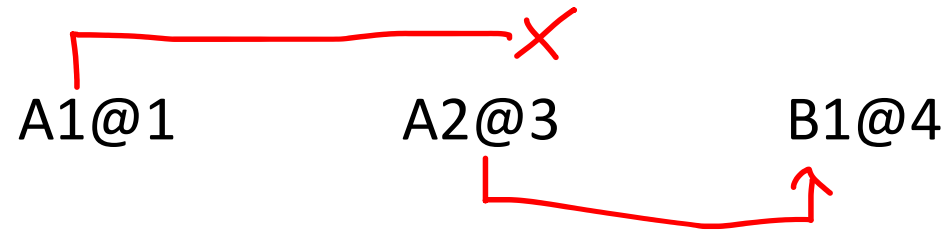
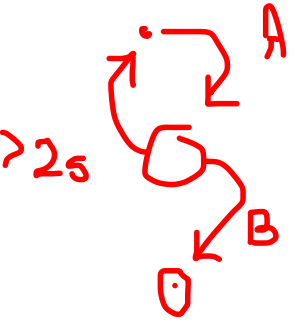
Pattern matching



Pattern	Results
every A -> B	{A1, B1}, {A2, B1}
every A -> (B and not A)	{A2, B1}

The *and not* operator causes the sub-expression looking for {A1, B?} to end when A2 arrives

Pattern matching



Pattern	Results
every A -> B	{A1, B1}, {A2, B1}
every A -> (B where timer:within(2 sec))	{A2, B1}

The *timer:within* operator causes the sub-expression looking for {A1, B?} to end after 2 seconds

Combine queries

- The insert into clause forwards events to other streams for further downstream processing

```
insert      into FireComplexEvent
select      a.sensor as sensor,
            a.smoke as smoke,
            b.temperature as temperature
from        pattern
            [every a=SmokeSensorEvent(smoke=true)
            -> every
            b=TemperatureSensorEvent(
            sensor=a.sensor, temperature>50)];

select      count(*)
from        FireComplexEvent.win:time(10 min);
```

Reference

- Esper documentation online
 - https://esper.espertech.com/release-8.8.0/reference-esper/html_single/
- Esper EPL online
 - <http://esper-epl-tryout.appspot.com/epltryout/mainform.html>

License and Acknowledgment

- This work is licensed under the Creative Commons Attribution-ShareAlike International Public License
- The original slides were prepared by Alessandro Margara for the PhD course on “Stream and Complex Event Processing in the Big Data Era” offered in 2019

<http://streamreasoning.org/events/scep2019>

Vertically Scalable solutions illustrated via EPL and Esper

Emanuele Della Valle

Politecnico di Milano