

Streaming Dataflow Model

Terminology: What is streaming?

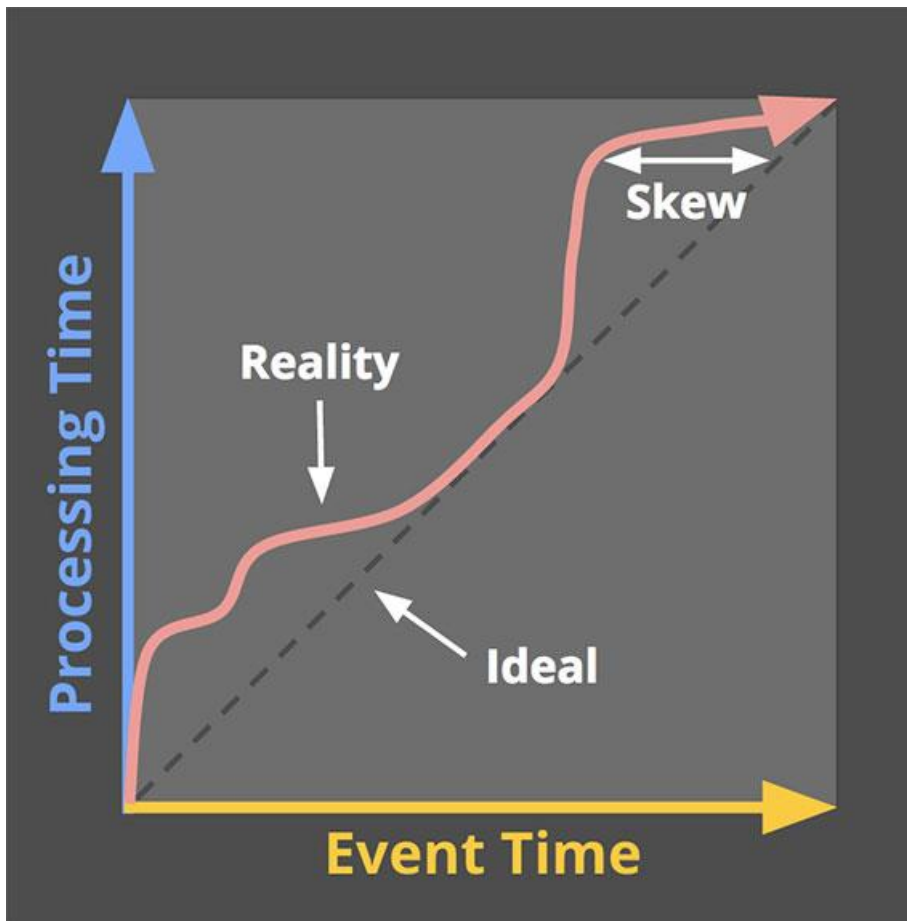
- **Unbounded data:** A type of ever-growing, essentially infinite data set. These are often referred to as “streaming data.”
- **Unbounded data processing:** An ongoing model of data processing, applied to the aforementioned type of unbounded data.
- **Low-latency, approximate, and/or speculative results:** These types of results are most often associated with streaming engines.

Event time vs. processing time

Within any data processing system, there are typically two domains of time we care about:

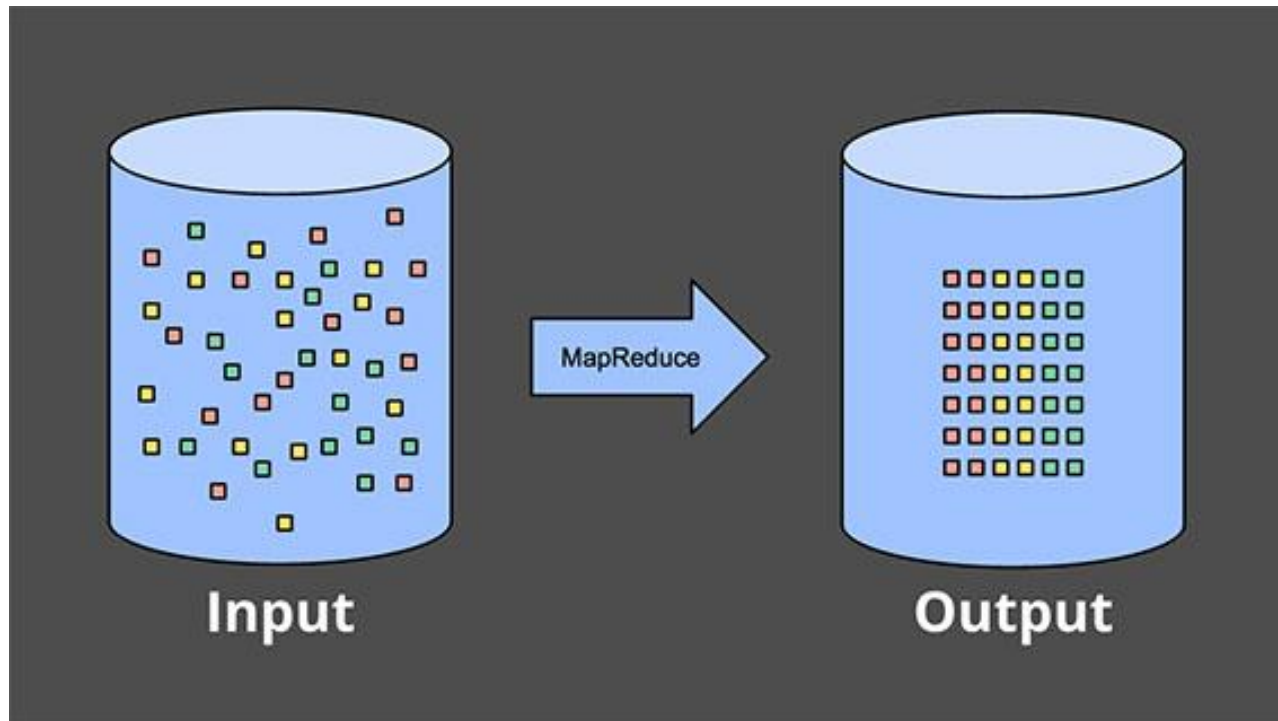
- **Event time**, which is the time at which events actually *occurred*.
- **Processing time**, which is the time at which events are *observed* in the system.

Example time domain mapping



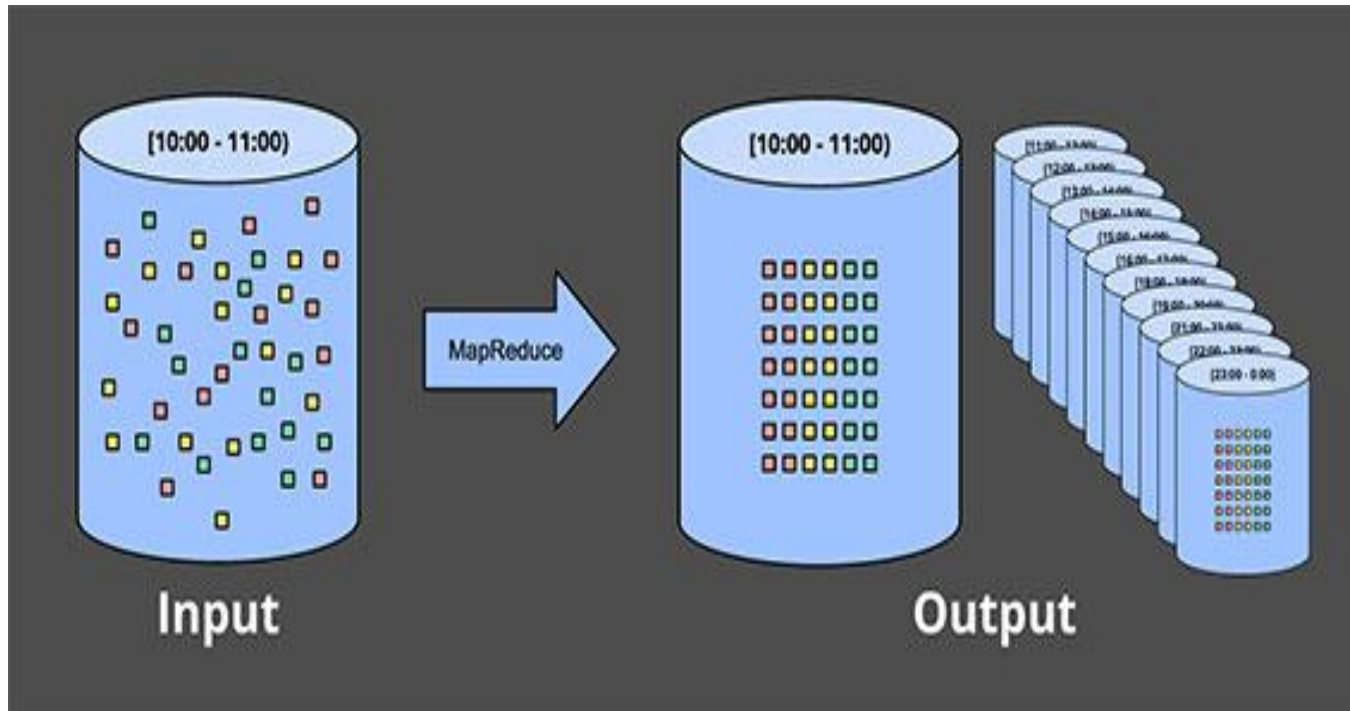
The black dashed line with a slope of one represents the ideal, where processing time and event time are exactly equal; the red line represents reality.

Bounded data



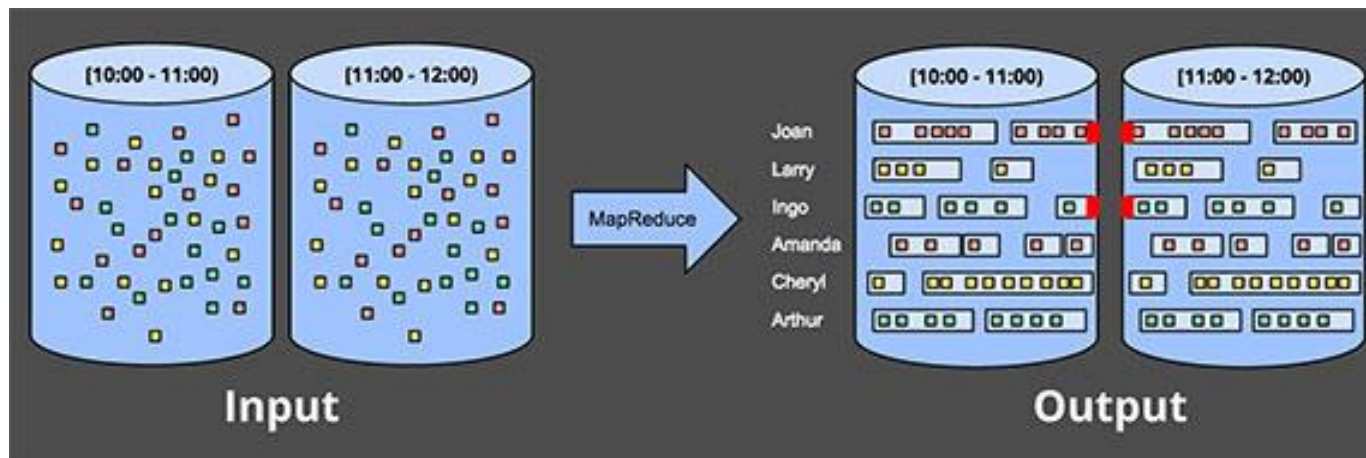
We run it through some data processing engine (typically batch, though a well-designed streaming engine would work just as well), such as [MapReduce](#), and on the right end up with a new structured data set with greater inherent value

Fixed windows



An unbounded data set is collected up front into finite, fixed-size windows of bounded data that are then processed via successive runs of a classic batch engine.

Sessions



An unbounded data set is collected up front into finite, fixed-size windows of bounded data that are then subdivided into dynamic session windows via successive runs a of classic batch engine.

Unbounded data — streaming

For many real-world, distributed input sources, you not only find yourself dealing with unbounded data, but also data that are:

- **Highly unordered with respect to event times**, meaning you need some sort of time-based shuffle in your pipeline if you want to analyze the data in the context in which they occurred.
- **Of varying event time skew**, meaning you can't just assume you'll always see most of the data for a given event time X within some constant epsilon of time Y .

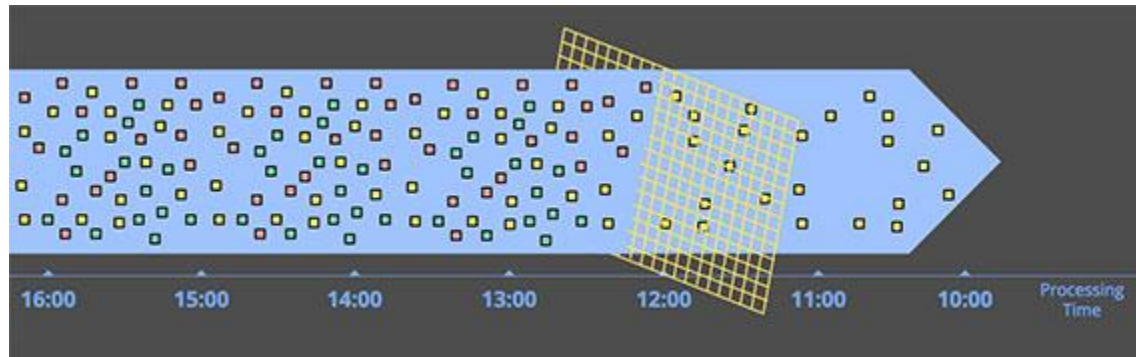
Four Approaches

- Time-agnostic
- Approximation
- Windowing by processing time
- Windowing by event time

Time-agnostic

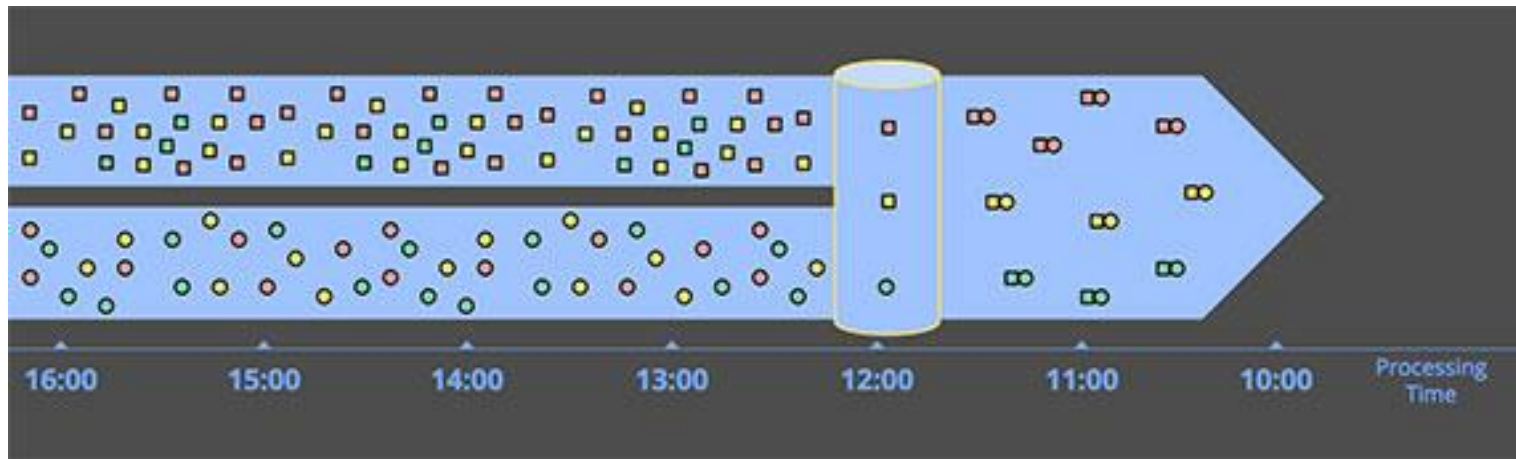
time-agnostic processing is used in cases where time is essentially irrelevant — i.e., all relevant logic is data driven. Since everything about such use cases is dictated by the arrival of more data, there's really nothing special a streaming engine has to support other than basic data delivery.

Filtering



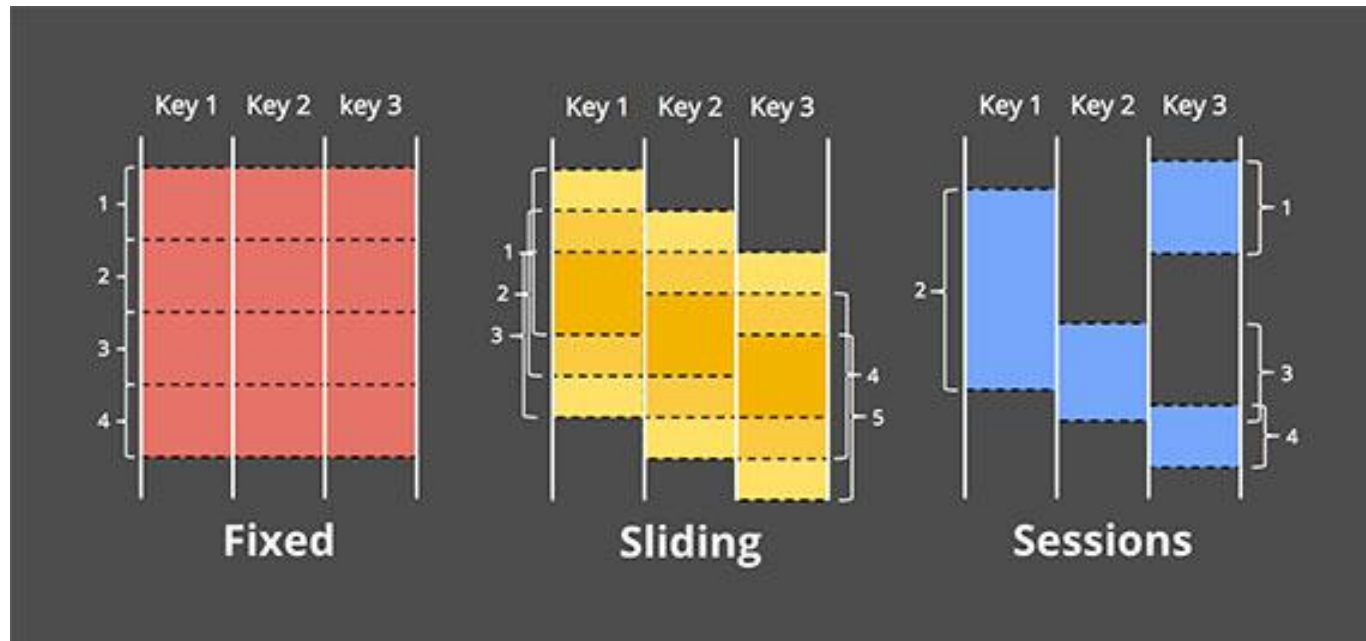
A very basic form of time-agnostic processing is filtering

Inner-joins



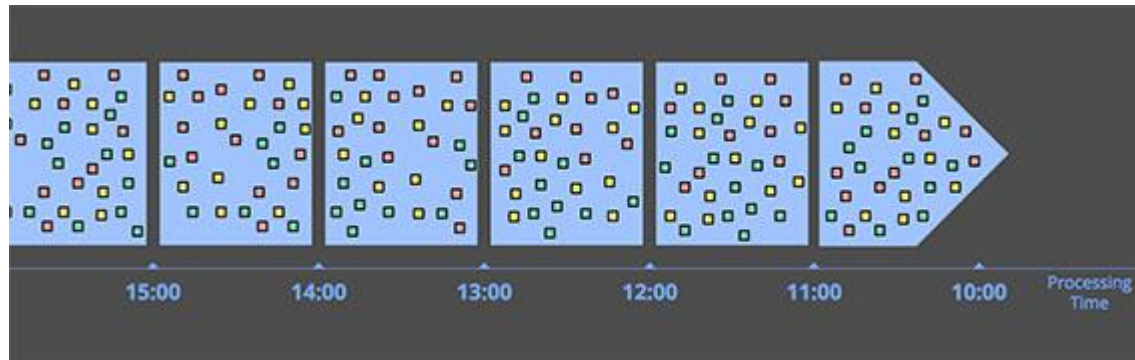
Another time-agnostic example is an inner-join (or hash-join). When joining two unbounded data sources, if you only care about the results of a join when an element from both sources arrive, there's no temporal element to the logic

Windowing



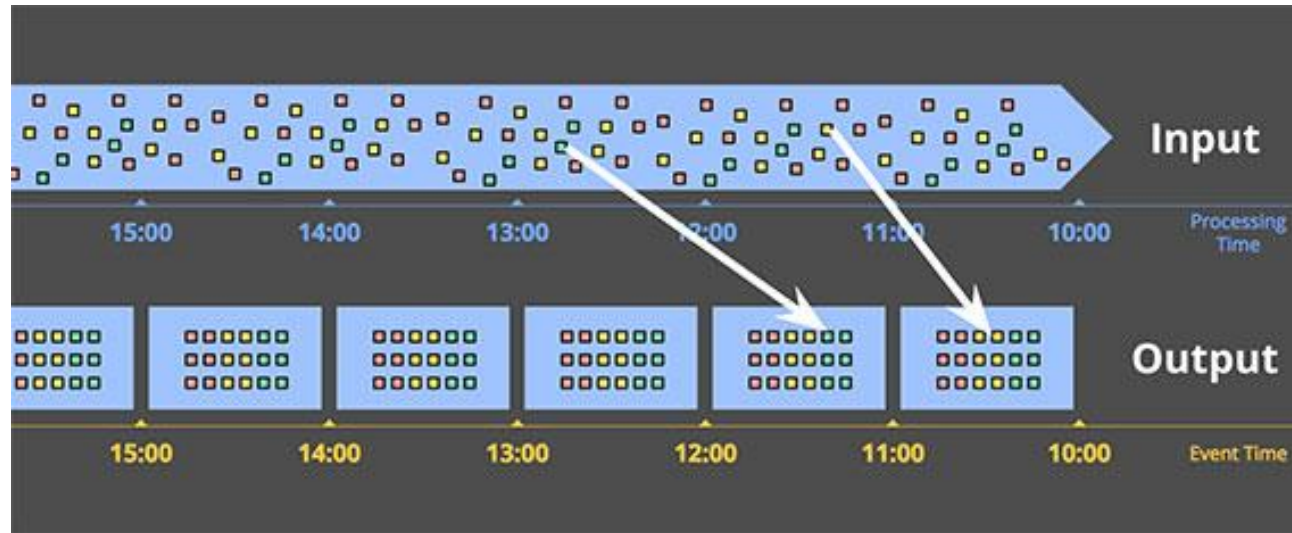
Windowing is simply the notion of taking a data source (either unbounded or bounded), and chopping it up along temporal boundaries into finite chunks for processing.

Windowing by processing time



When windowing by processing time, the system essentially buffers up incoming data into windows until some amount of processing time has passed.

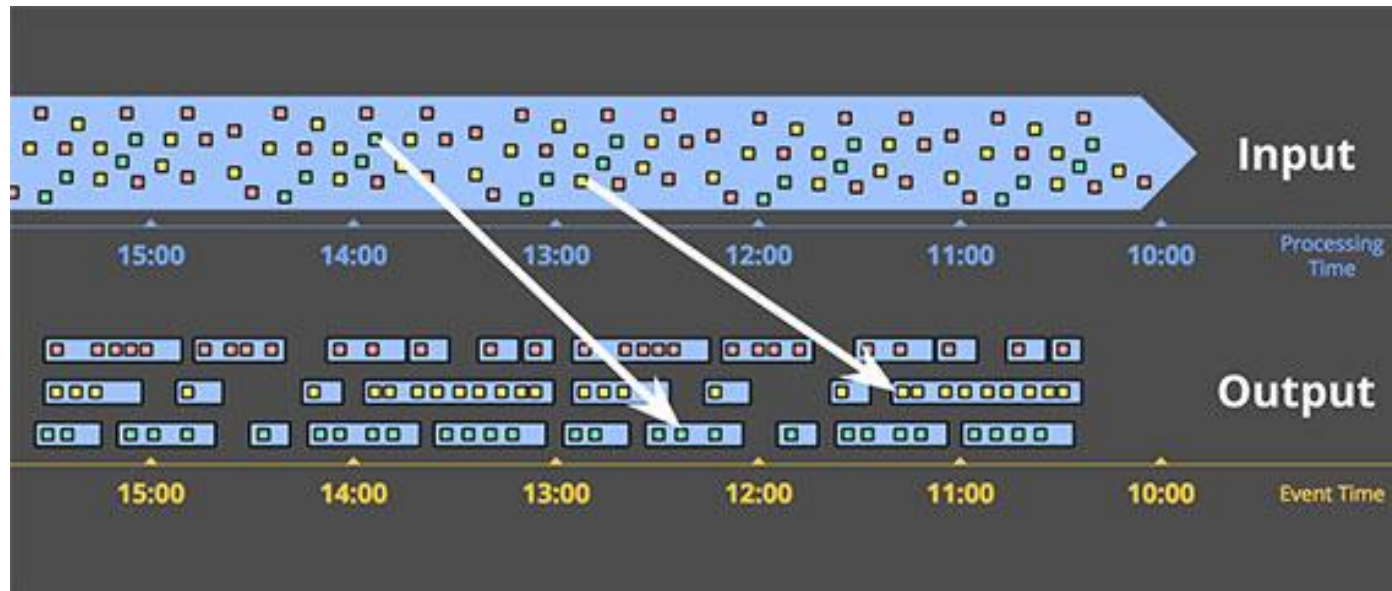
Windowing by event time



Event time windowing is what you use when you need to observe a data source in finite chunks that reflect the times at which those events actually happened. It's the gold standard of windowing.

Sadly, most data processing systems in use today lack native support for it.

Windowing into session windows by event time



Another nice thing about event time windowing over an unbounded data source is that you can create dynamically sized windows, such as sessions, without the arbitrary splits observed when generating sessions over fixed windows

Event time windows have two notable drawbacks

- **Buffering:** Due to extended window lifetimes, more buffering of data is required.
- **Completeness:** Given that we often have no good way of knowing when we've seen *all* the data for a given window, how do we know when the results for the window are ready to materialize? In truth, we simply don't.

Conclusion I

- Established the important **differences between event time and processing time**, characterized the **difficulties those differences impose** when analyzing data in the context of when they occurred, and **proposed a shift in approach away from notions of completeness and toward simply adapting to changes** in data over time.
- Looked at the **major data processing approaches** in common use today for bounded and unbounded data, via both batch and streaming engines, roughly categorizing the unbounded approaches into: **time-agnostic, approximation, windowing by processing time**, and **windowing by event time**.

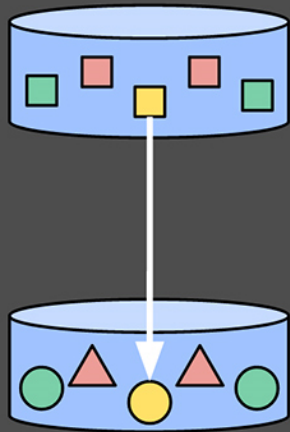
More concepts

- **Watermarks:** A watermark is a notion of input completeness with respect to event times. A watermark with a value of time X makes the statement: “all input data with event times less than X have been observed.”
- **Triggers:** A trigger is a mechanism for declaring when the output for a window should be materialized relative to some external signal. Triggers provide flexibility in choosing when outputs should be emitted.
- **Accumulation:** An accumulation mode specifies the relationship between multiple results that are observed for the same window. Those results might be completely disjointed, i.e., representing independent deltas over time, or there may be overlap between them.

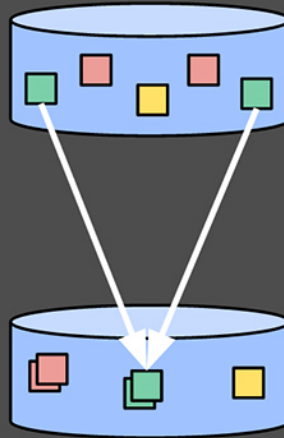
What/Where/When/How

- **What results are calculated?** This question is answered by the types of *transformations* within the pipeline.
- **Where in event time are results calculated?** This question is answered by the use of *event-time windowing* within the pipeline.
- **When in processing time are results materialized?** This question is answered by the use of watermarks and triggers.
- **How do refinements of results relate?** This question is answered by the type of *accumulation* used: discarding (where results are all independent and distinct), accumulating (where later results build upon prior ones), or accumulating and retracting (where both the accumulating value plus a retraction for the previously triggered value(s) are emitted).

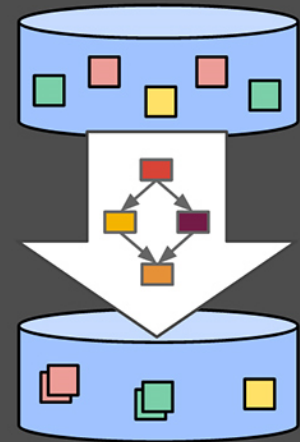
What: transformations



Element-Wise

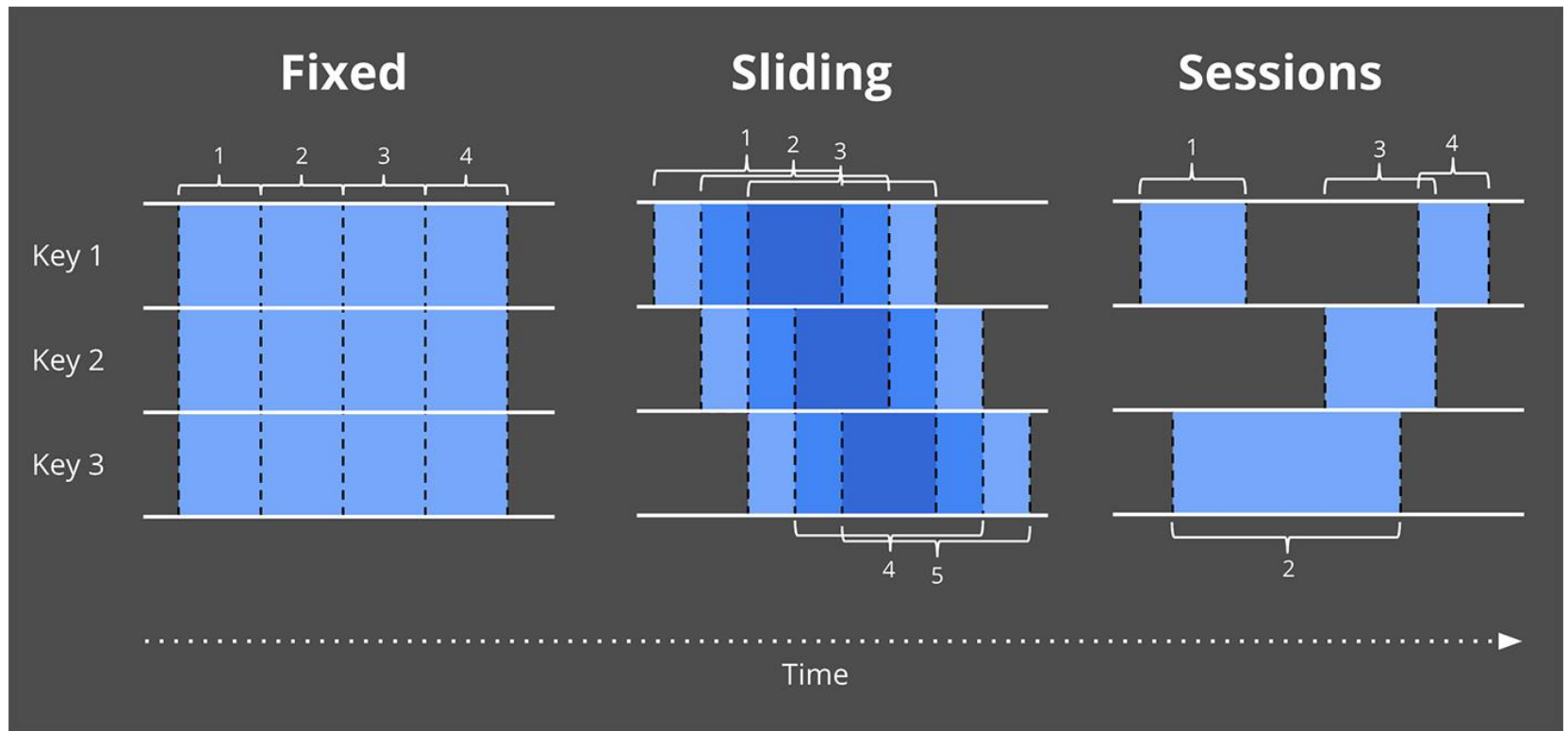


Aggregating



Composite

Where: windowing

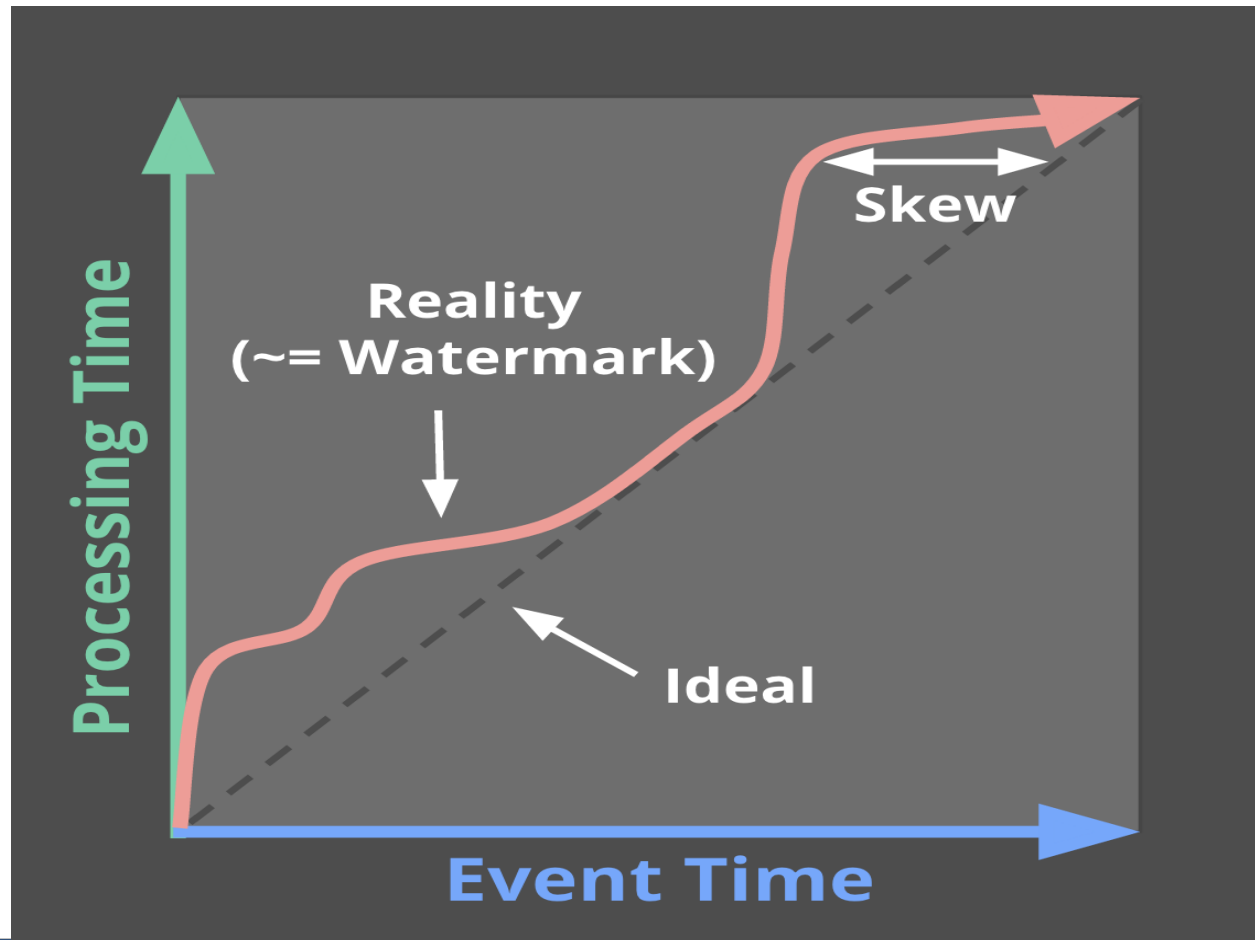


When: watermarks

Watermarks are the first half of the answer to the question: “***When* in processing time are results materialized?**” Watermarks are temporal notions of input completeness in the event-time domain.

Conceptually, you can think of the watermark as a function, $F(P) \rightarrow E$, which takes a point in processing time and returns a point in event time.

Event time progress, skew, and watermarks

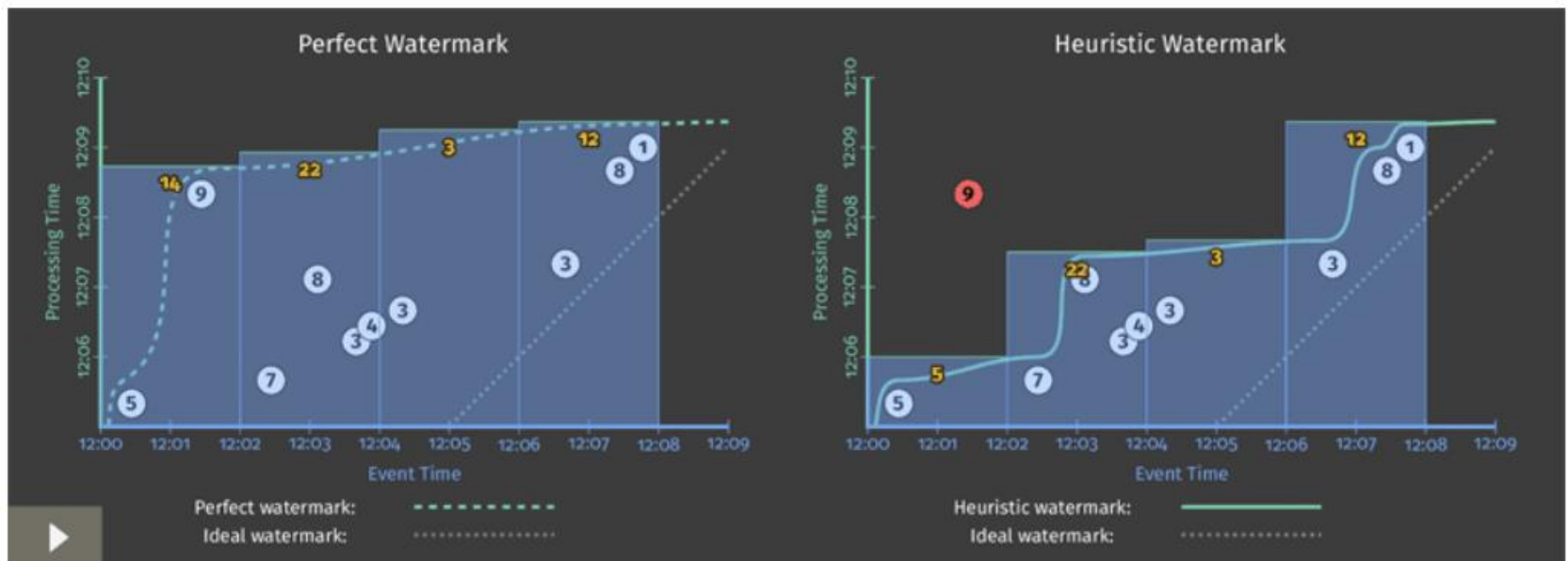


Watermarks

Perfect watermarks: In the case where we have perfect knowledge of all of the input data, it's possible to construct a perfect watermark; in such a case, there is no such thing as late data; all data are early or on time.

Heuristic watermarks: For many distributed input sources, perfect knowledge of the input data is impractical, in which case the next best option is to provide a heuristic watermark.

Watermarks



That point in event time, E , is the point up to which the system believes all inputs with event times less than E have been observed. In other words, it's an assertion that no more data with event times less than E will ever be seen again.

Two shortcomings of watermarks

Too slow: When a watermark of any type is correctly delayed due to known unprocessed data (e.g., slowly growing input logs due to network bandwidth constraints), that translates directly into delays in output if advancement of the watermark is the only thing you depend on for stimulating results.

Too fast: When a heuristic watermark is incorrectly advanced earlier than it should be, it's possible for data with event times before the watermark to arrive some time later, creating late data.

Addressing these shortcomings is where triggers come into play

Triggers are the second half of the answer to the question:
“***When* in processing time are results materialized?**”

Triggers declare when output for a window should happen in processing time.

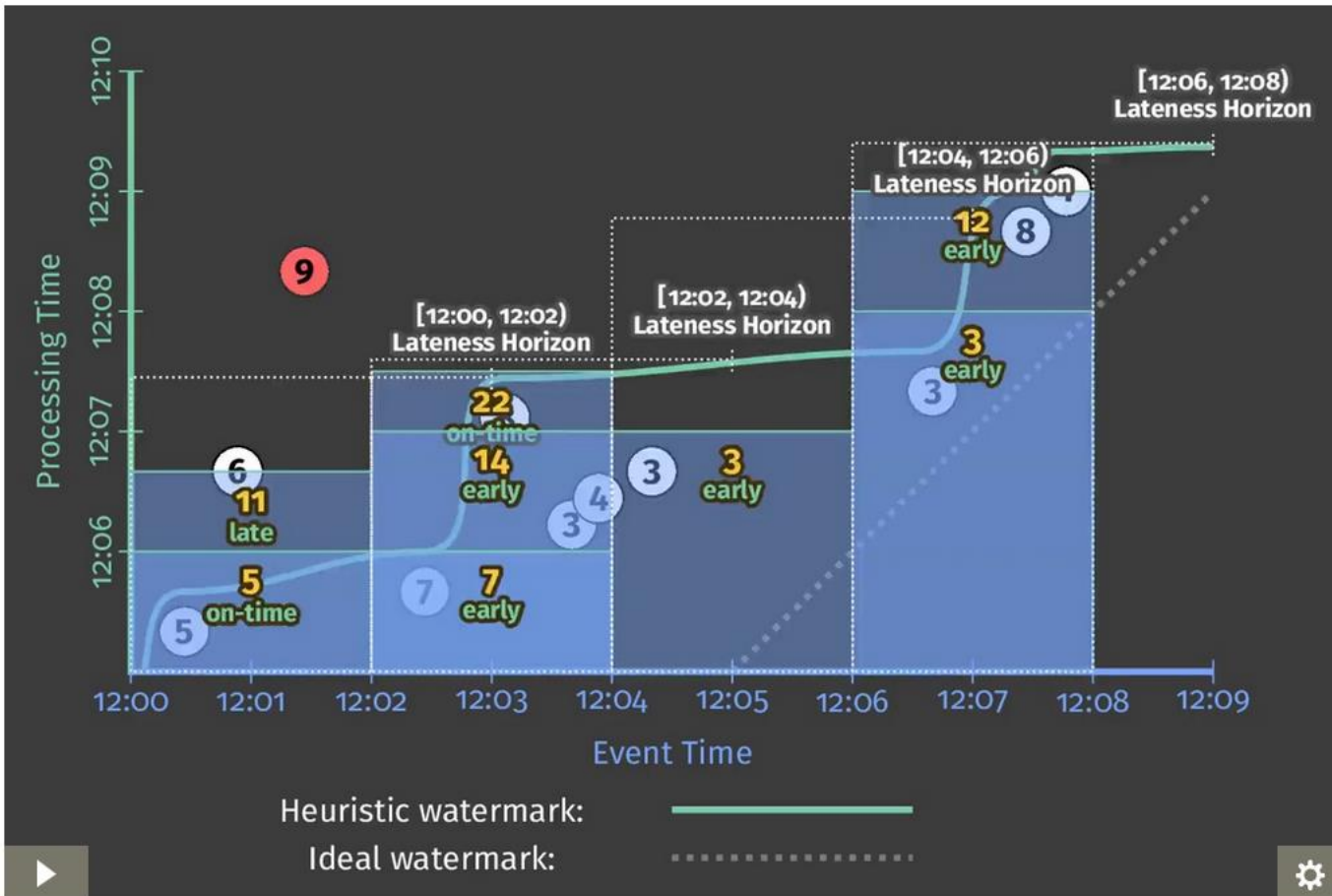
Examples of signals used for triggering

- **Watermark progress (i.e., event time progress)**, an implicit version of which we already saw in Figure, where outputs were materialized when the watermark passed the end of the window^l.
- **Processing time progress**, which is useful for providing regular, periodic updates since processing time (unlike event time) always progresses more or less uniformly and without delay.
- **Element counts**, which are useful for triggering after some finite number of elements have been observed in a window.
- **Punctuations**, or other data-dependent triggers, where some record or feature of a record (e.g., an EOF element or a flush event) indicates that output should be generated.

When: allowed lateness (i.e., garbage collection)

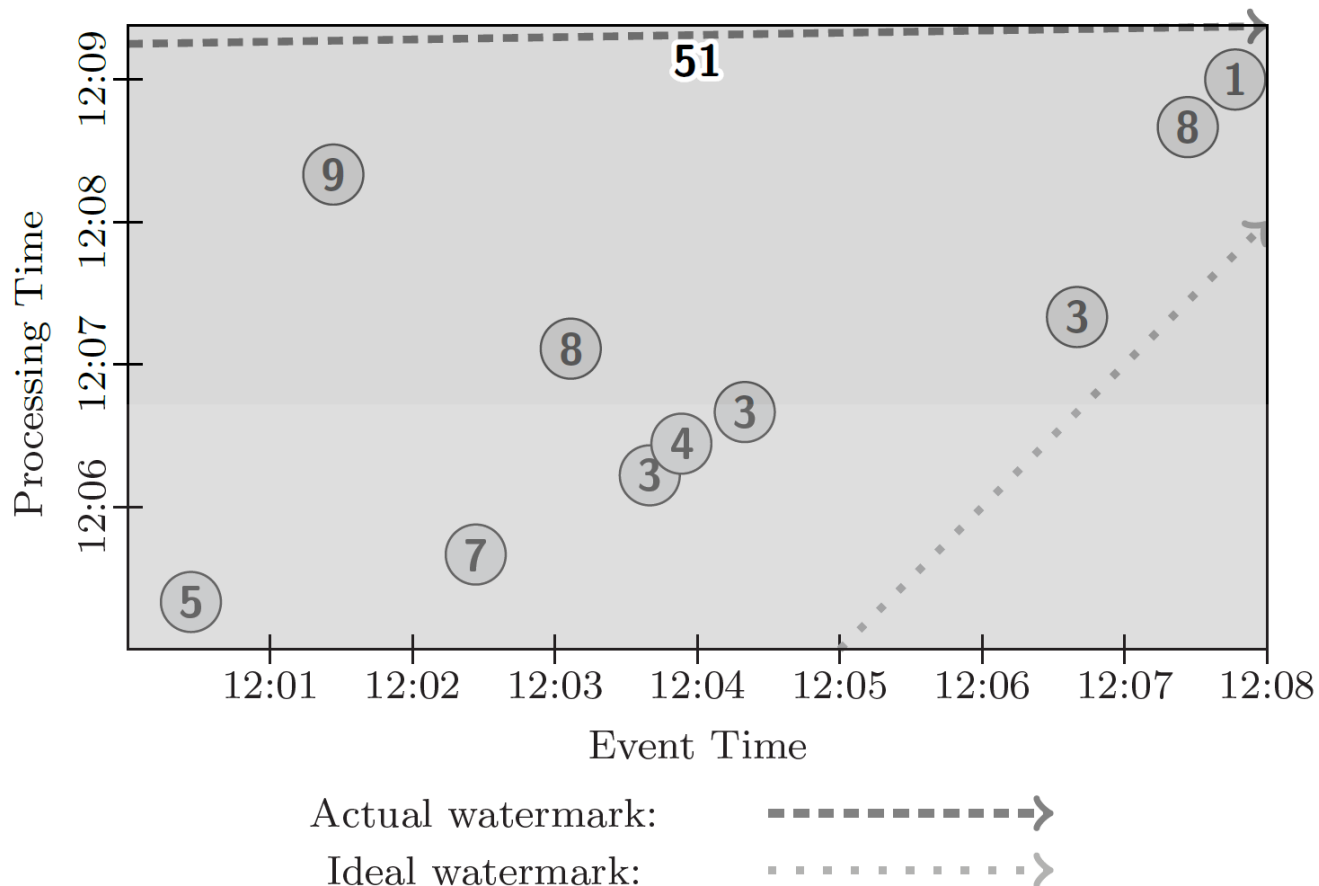
- Any real-world, out-of-order processing system needs to provide some way to bound the lifetimes of the windows it's processing.
- A clean and concise way of doing this is by defining a horizon on the allowed lateness within the system—i.e., placing a bound on how late any given *record* may be (relative to the watermark) for the system to bother processing it; any data that arrive after this horizon are simply dropped.

Lateness horizons

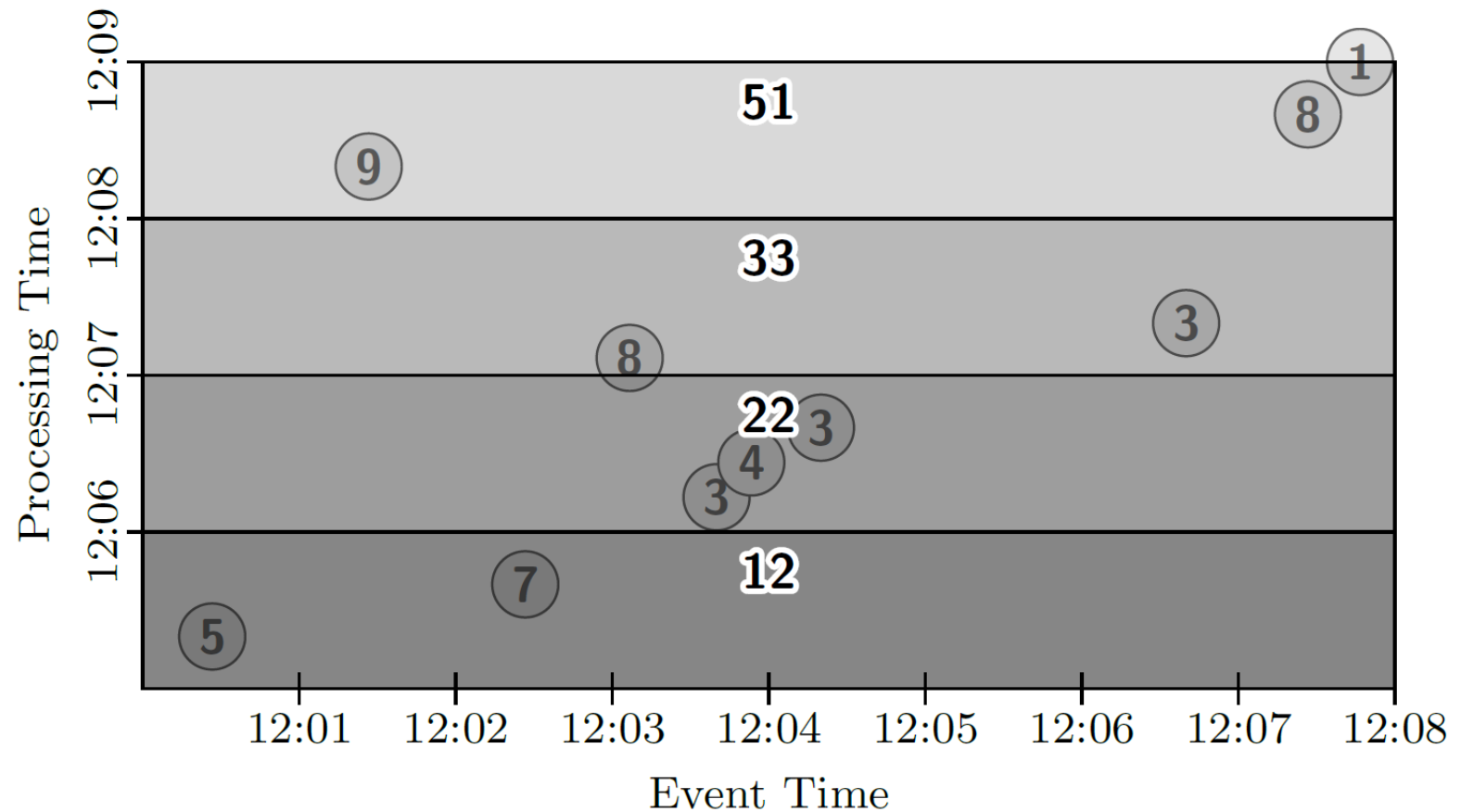


Examples

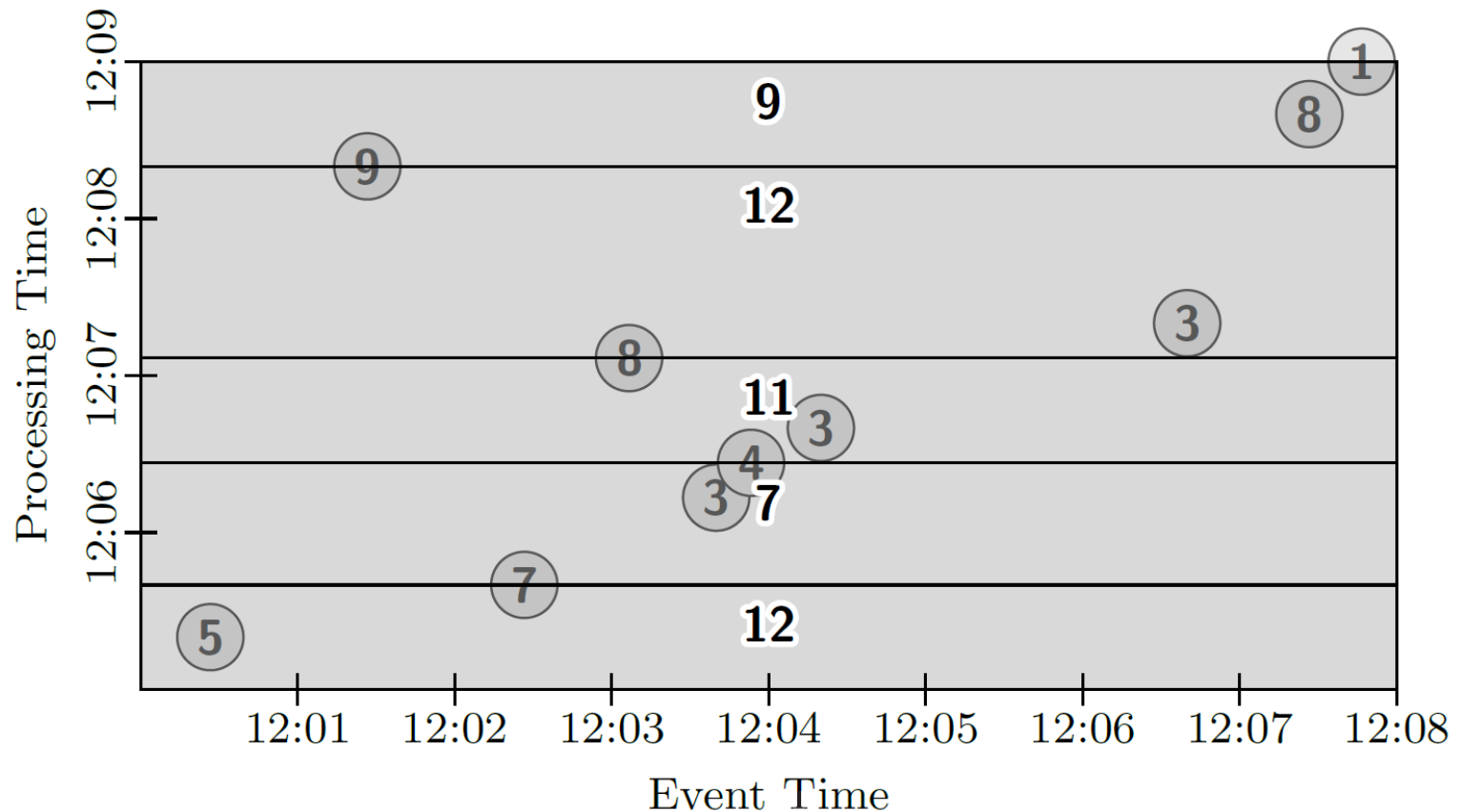
Classic Batch Processing



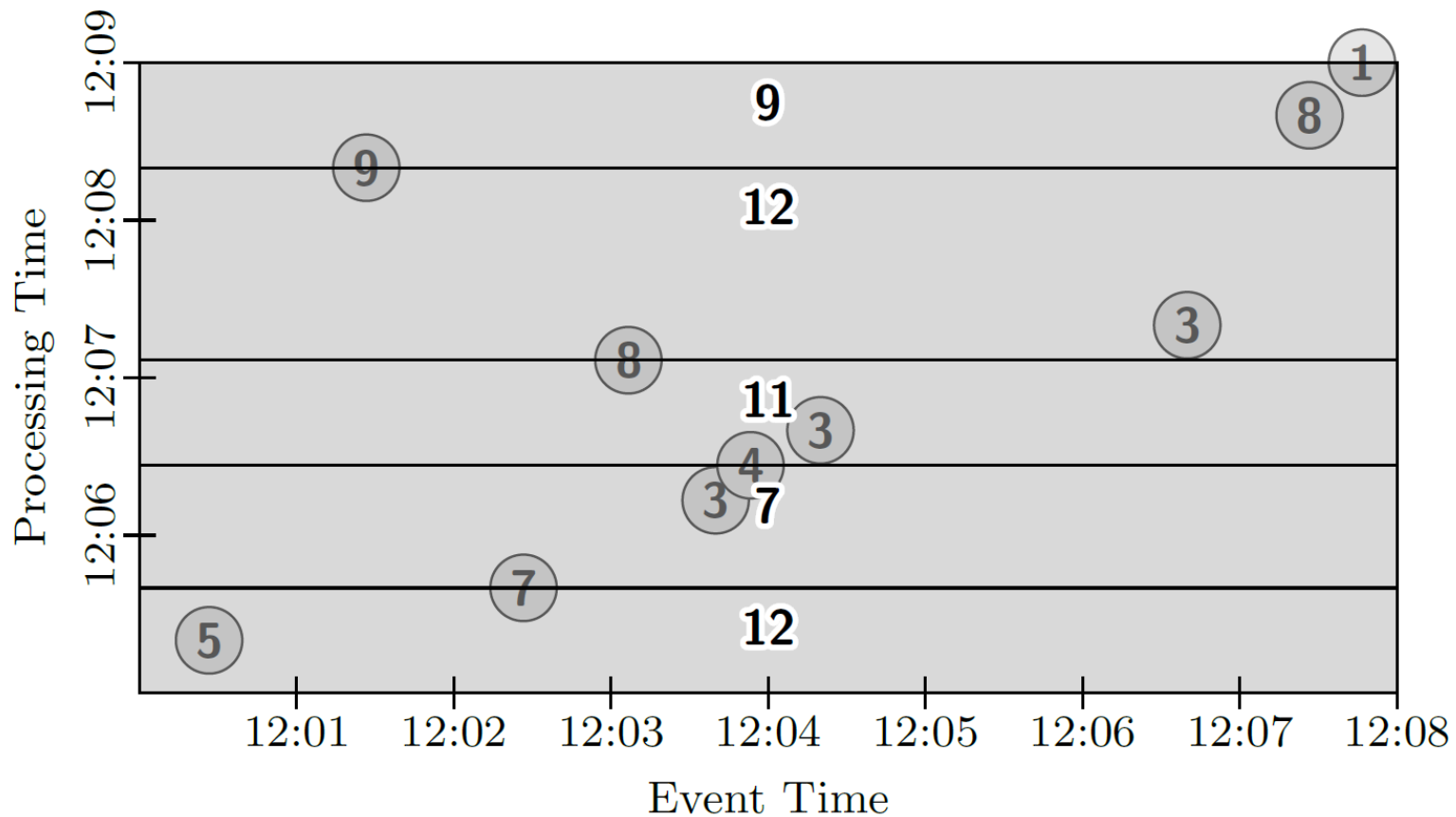
GlobalWindows, AtPeriod, Accumulating



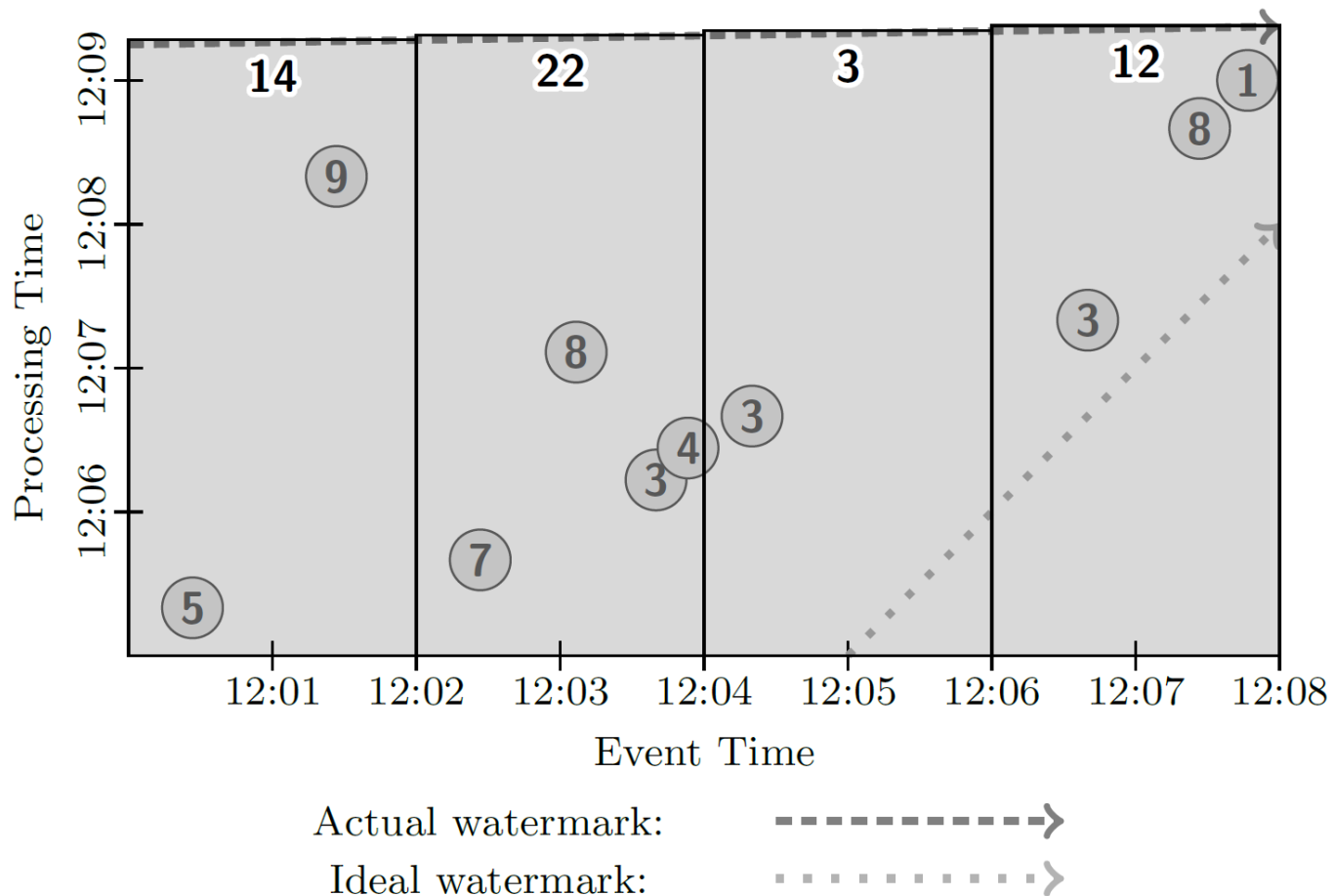
GlobalWindows, AtCount, Discarding



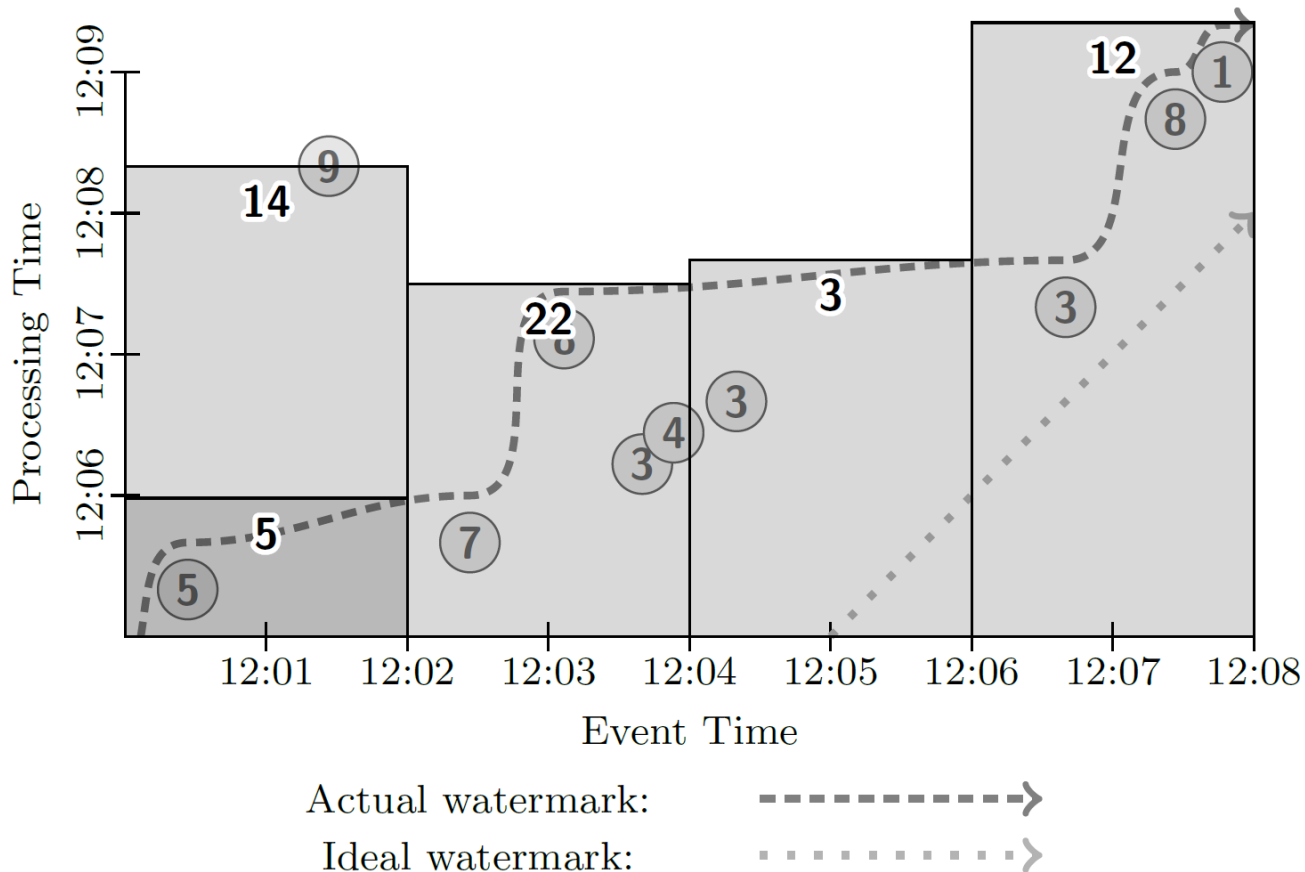
GlobalWindows, AtCount, Discarding



FixedWindows, Batch



FixedWindows, Streaming



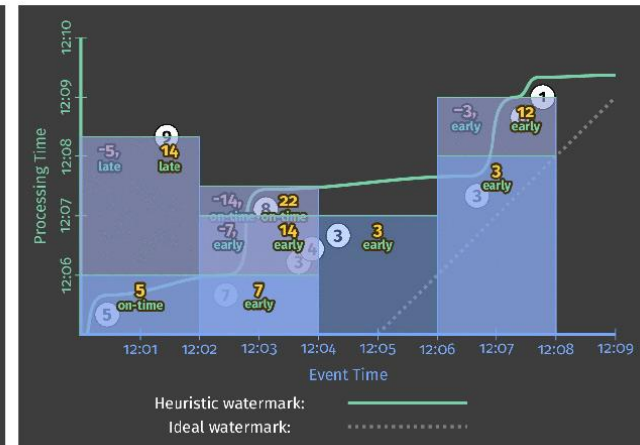
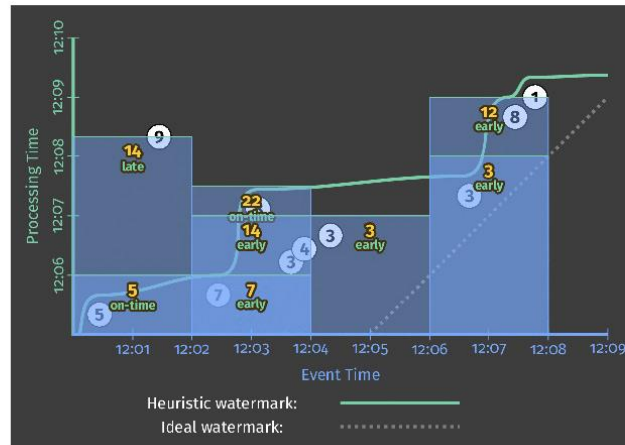
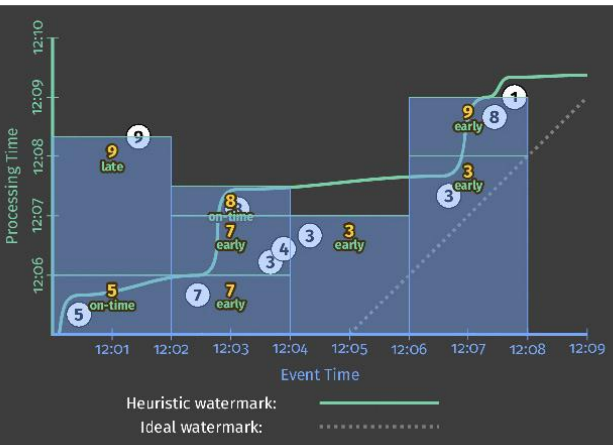
How: accumulation

- **Discarding:** Every time a pane is materialized, any stored state is discarded. This means each successive pane is independent from any that came before.
- **Accumulating:** As in every time a pane is materialized, any stored state is retained, and future inputs are accumulated into the existing state.
- **Accumulating & retracting:** Like accumulating mode, but when producing a new pane, also produces independent retractions for the previous pane(s)

Examples

	Discarding	Accumulating	Accumulating & Retracting
Pane 1: [7]	7	7	7
Pane 2: [3, 4]	7	14	14, -7
Pane 3: [8]	8	22	22, -14
Last Value Observed	8	22	22
Total Sum	22	51	22

Examples



Side-by-side comparison of accumulation modes:
discarding (left), accumulating (center), and accumulating & retracting (right).

Conclusion II

What results are calculated? Answered via transformations.

Where in event time are results calculated? Answered via windowing.

When in processing time are results materialized? Answered via watermarks and triggers.

How do refinements of results relate? Answered via accumulation modes.

Major concepts

Event-time versus processing-time: The all-important distinction between when events occurred and when they are observed by your data processing system.

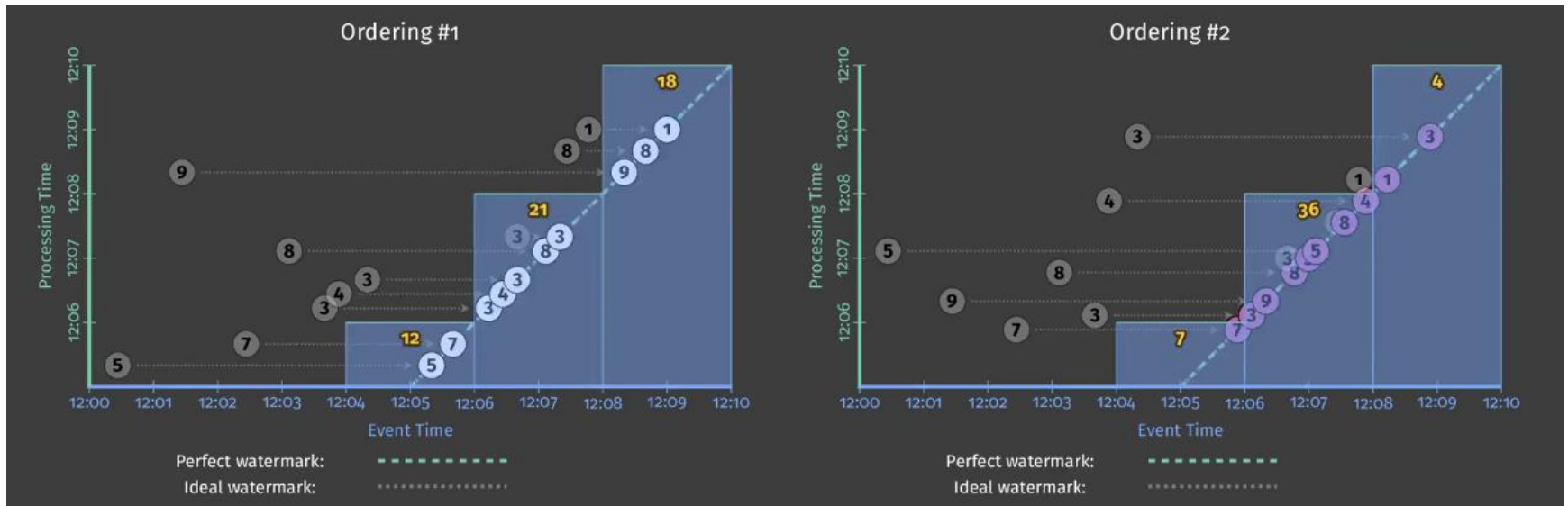
Windowing: The commonly utilized approach to managing unbounded data by slicing it up along temporal boundaries (in either processing-time or event-time, though **we narrow the definition of windowing in the Dataflow model to mean only within event-time**).

Watermarks: The powerful notion of progress in event-time that provides a means of reasoning about completeness in an out-of-order processing system operating on unbounded data.

Triggers: The declarative mechanism for specifying precisely when materialization of output makes sense for your particular use case.

Accumulation: The relationship between refinements of results for a single window in cases where it's materialized multiple times as it evolves.

Processing-time windowing via ingress time



Lastly, let's look at processing-time windowing achieved by mapping the event times of input data to be their ingress times.

Processing-time windowing via ingress time

- **Time-shifting:** When elements arrive, their event times need to be overwritten with the time of ingress.
- **Windowing:** Return to using standard fixed event-time windowing.
- **Triggering:** Since ingress time affords the ability to calculate a perfect watermark, we can use the default trigger, which in this case implicitly fires exactly once when the watermark passes the end of the window.
- **Accumulation mode:** Since we only ever have one output per window, the accumulation mode is irrelevant.

Conclusion II

While it's interesting to see the different ways one can implement processing-time windowing, the big takeaway here is the one: **event-time windowing is order-agnostic**, at least in the limit (actual panes along the way may differ until the input becomes complete); **processing-time windowing is not**.

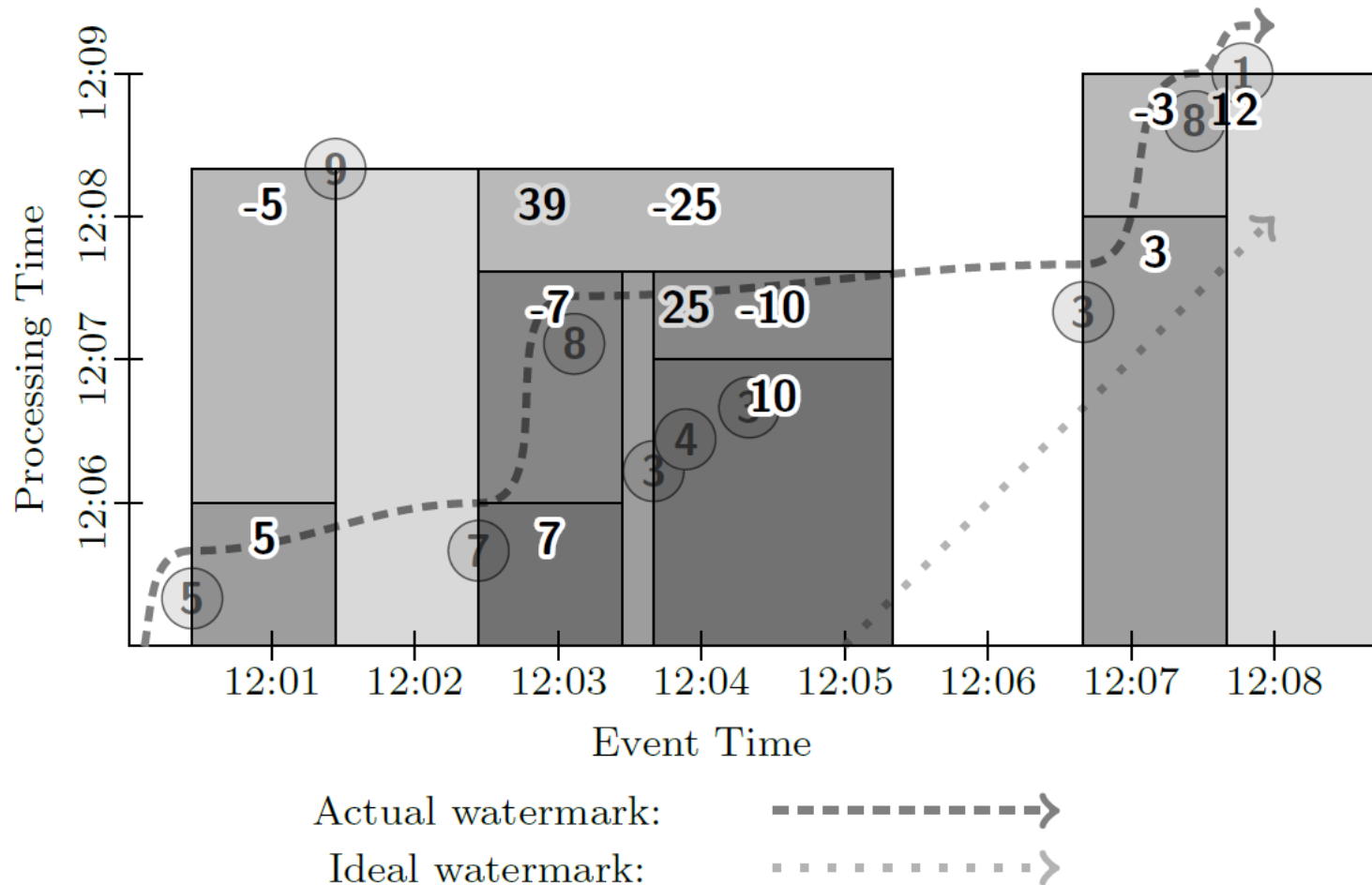
If you care about the times at which your events actually happened, you must use event-time windowing or your results will be meaningless.

Where: session windows

Sessions are a special type of window that captures a period of activity in the data that is terminated by a gap of inactivity.

They're particularly useful in data analysis because they can provide a view of the activities for a specific user over a specific period of time where they were engaged in some activity. This allows for the correlation of activities within the session, drawing inferences about levels of engagement based off of the lengths of the sessions, etc.

Session Retracting

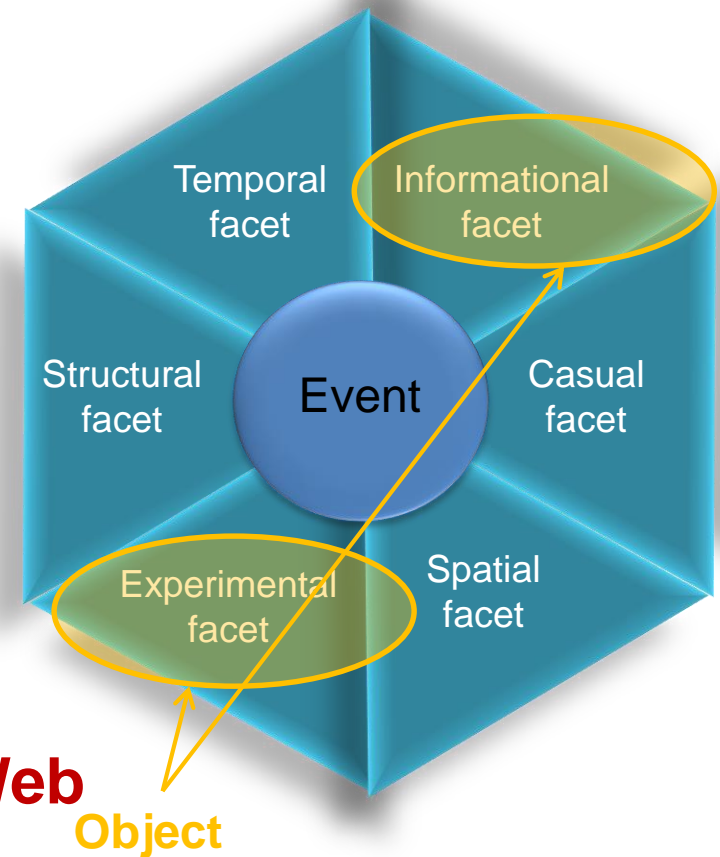


Real World Aware Applications

Event Driven Architecture

An event should explicitly define the following aspects:

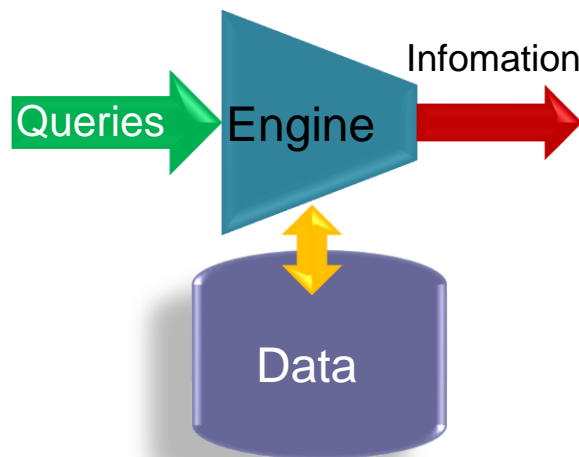
- information about event
- experiences related to event
- the event's structural and causal relationships with other events



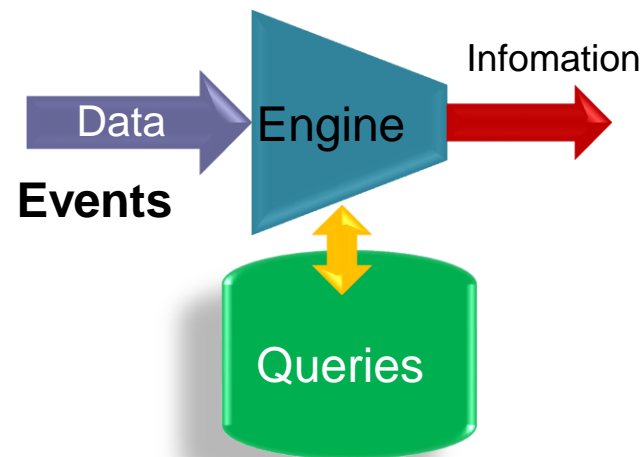
EventWeb – the next generation Web

Events Processing vs Data Processing

Upside-down philosophy

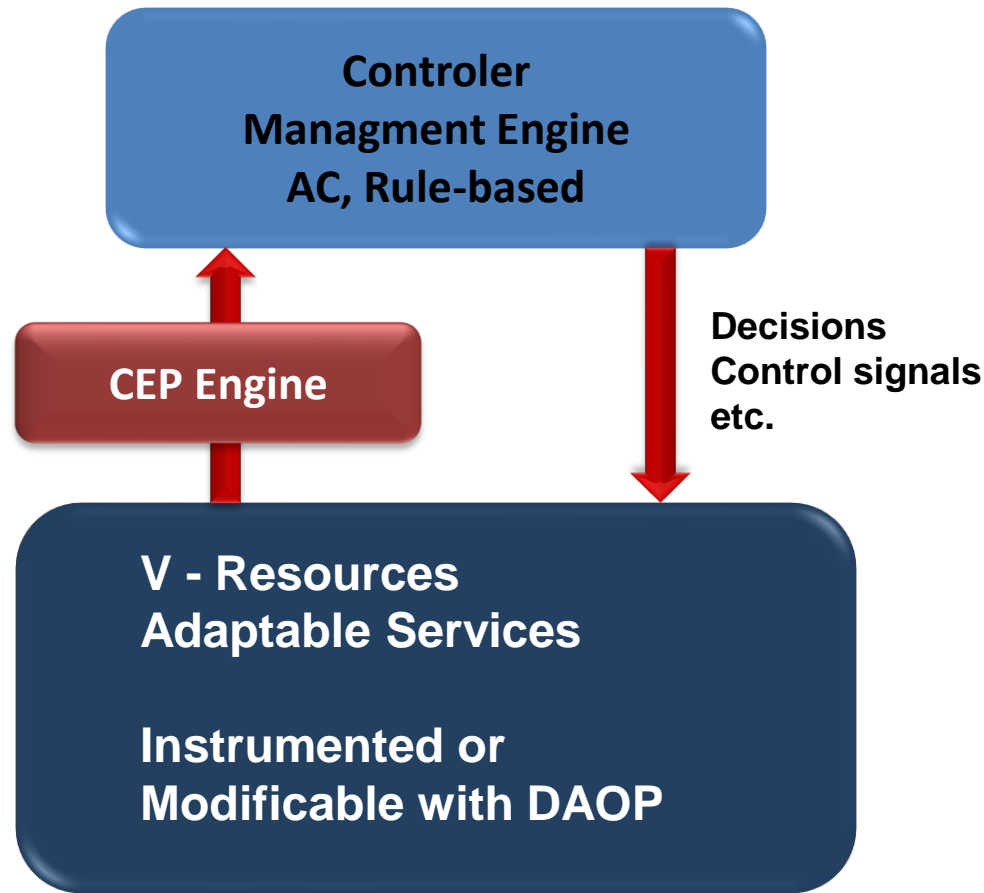


store data and run
the queries through

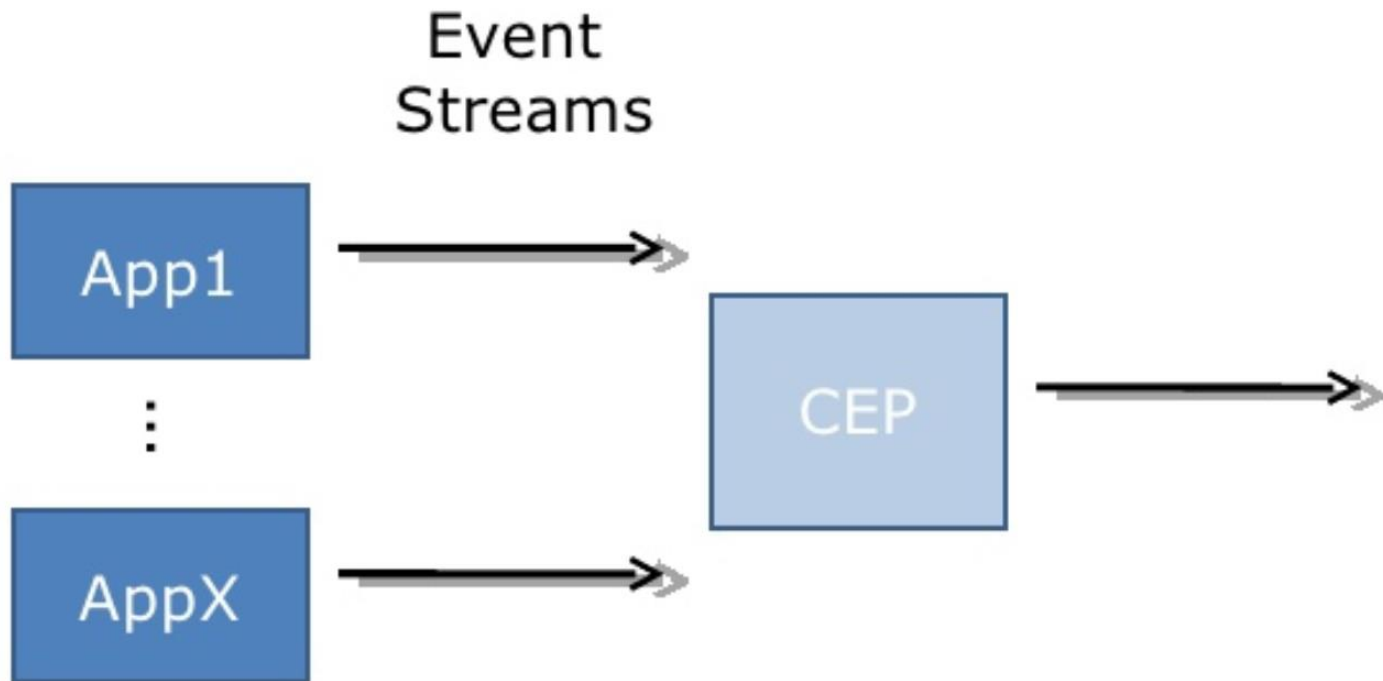


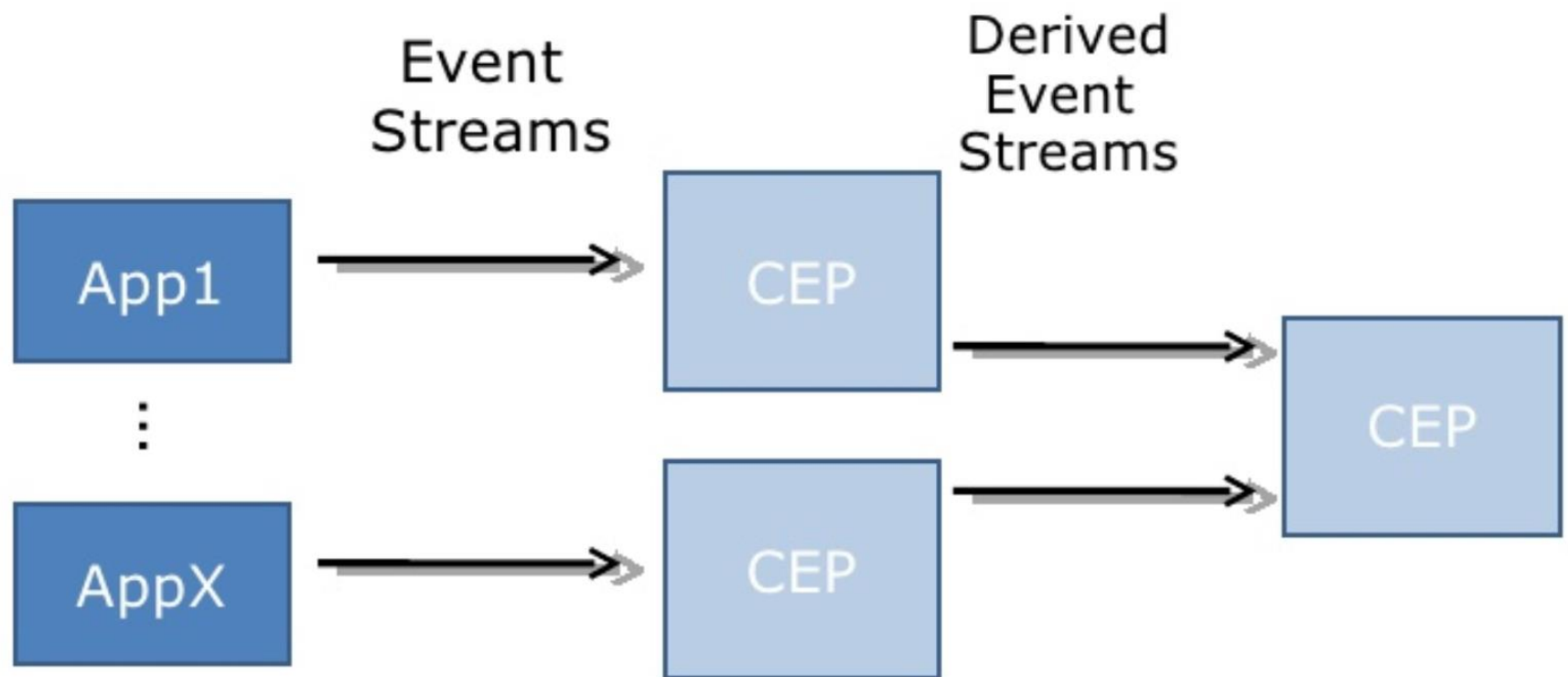
store queries and
run the data through

Large Picture

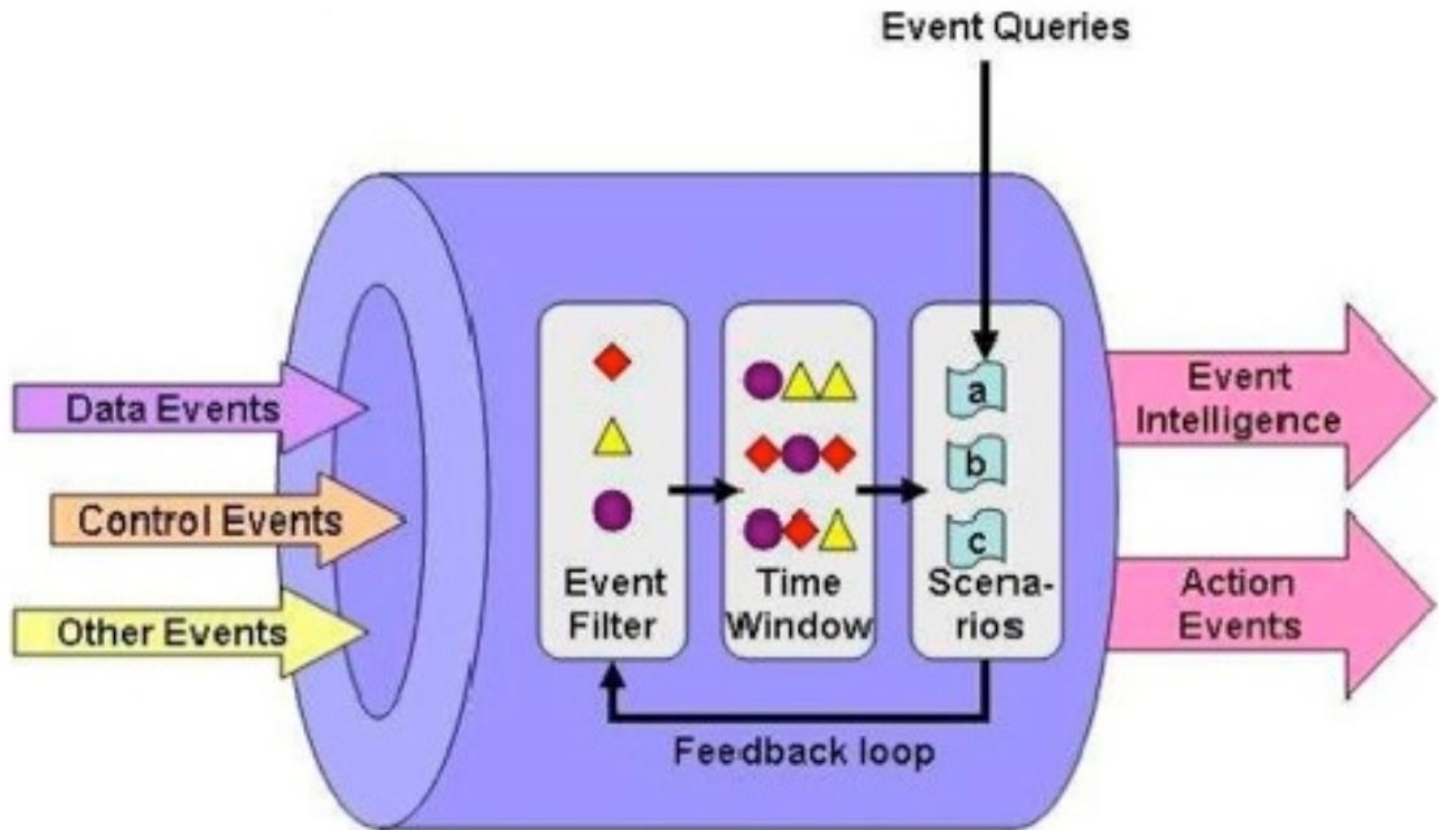


Complex Events Processing





EP Engine



Esper Engine



Open Source SOA – Jeff Davis

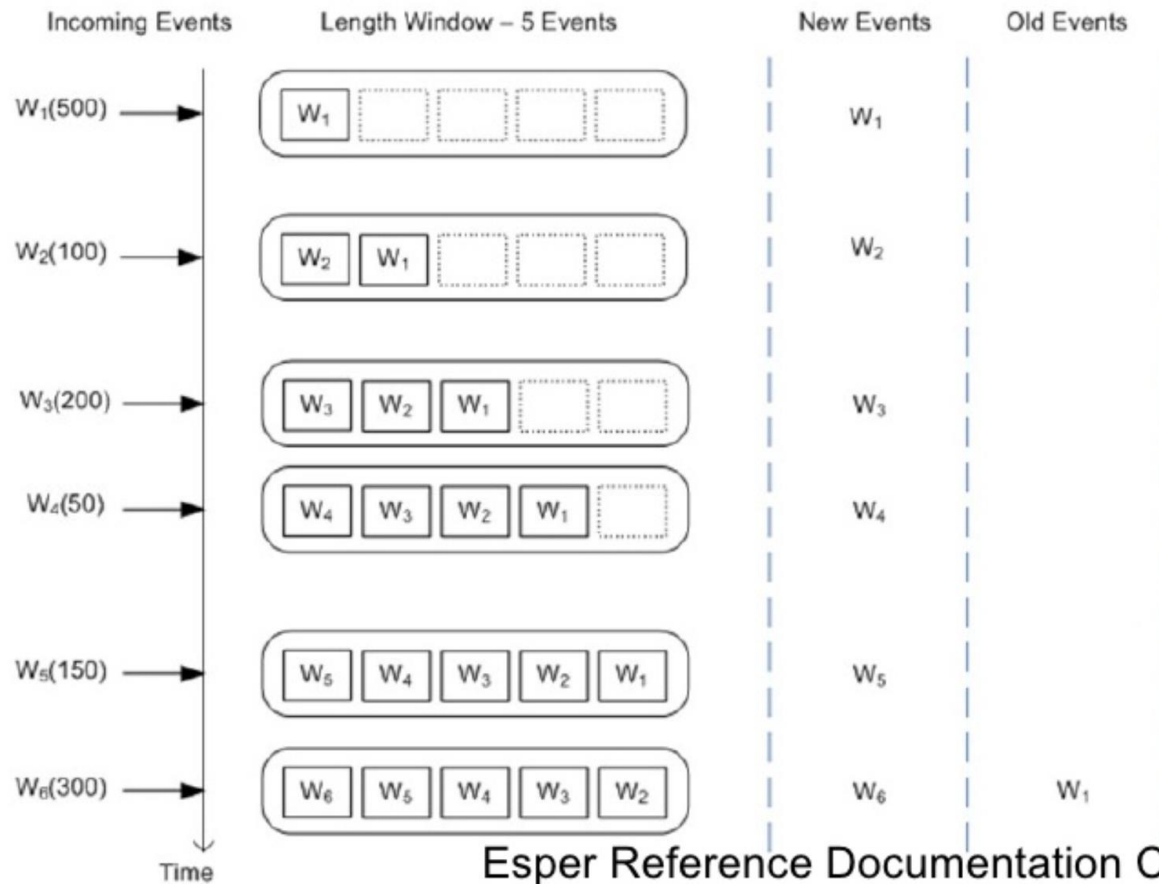
DB vs EP

SQL → EPL

Tables → Event Streams
Views

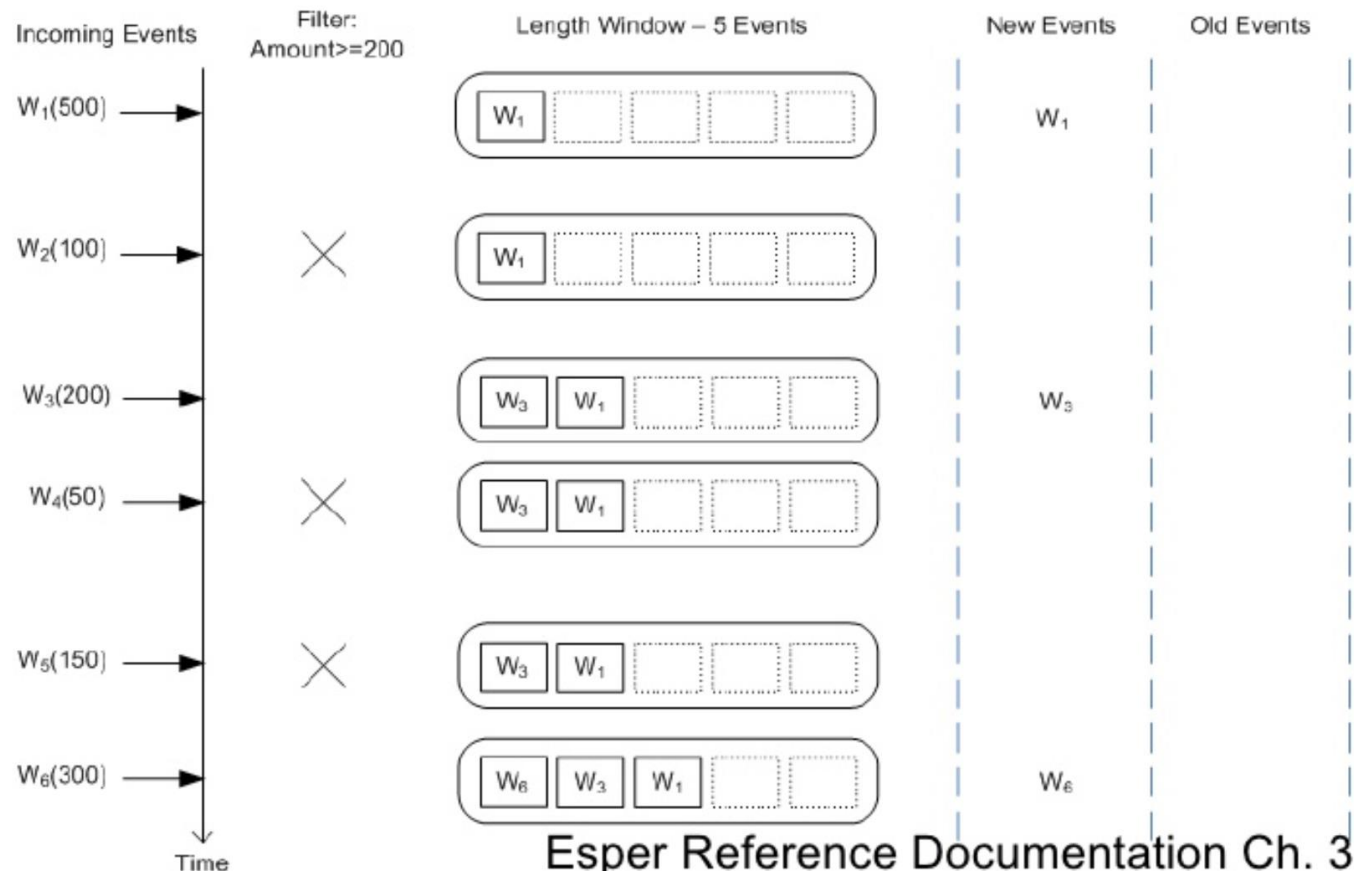
Rows → Events

select * from Withdrawal.win:length(5)



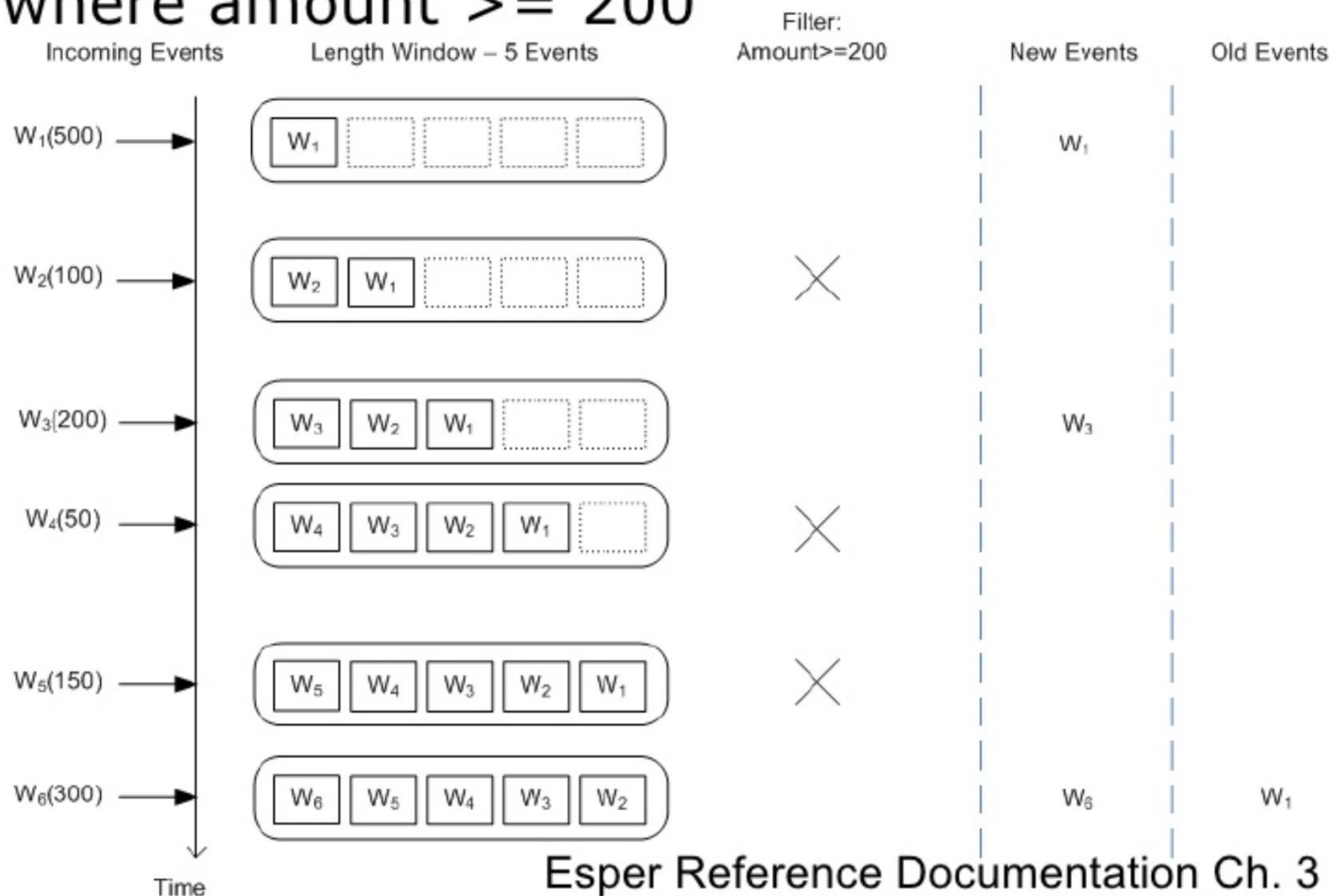
Esper Reference Documentation Ch. 3

select * from
Withdrawal(amount >= 200).win:length(5)



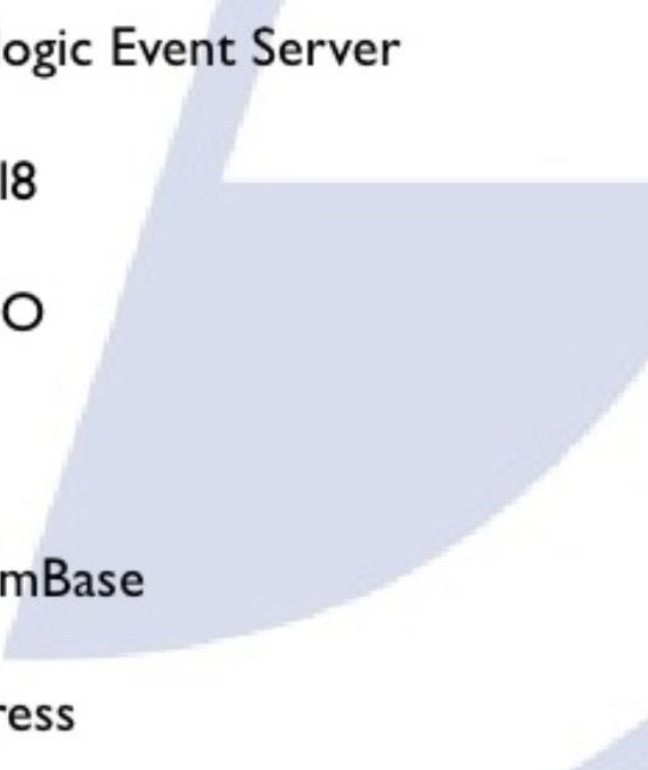
Esper Reference Documentation Ch. 3

select * from Withdrawal.win:length(5)
where amount >= 200



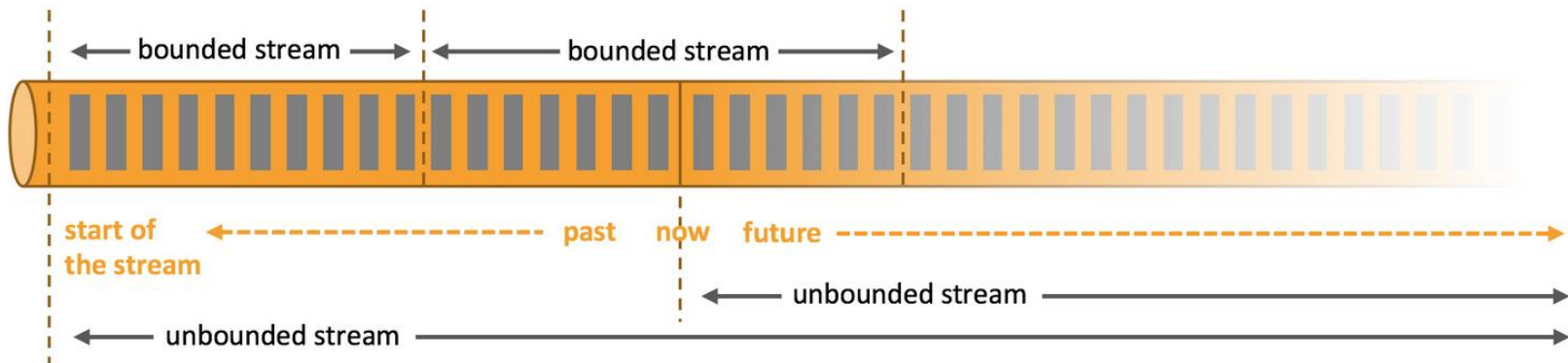
Esper Reference Documentation Ch. 3

EP Products

- Weblogic Event Server
 - Coral8
 - TIBCO
 - IBM
 - StreamBase
 - Progress
- 

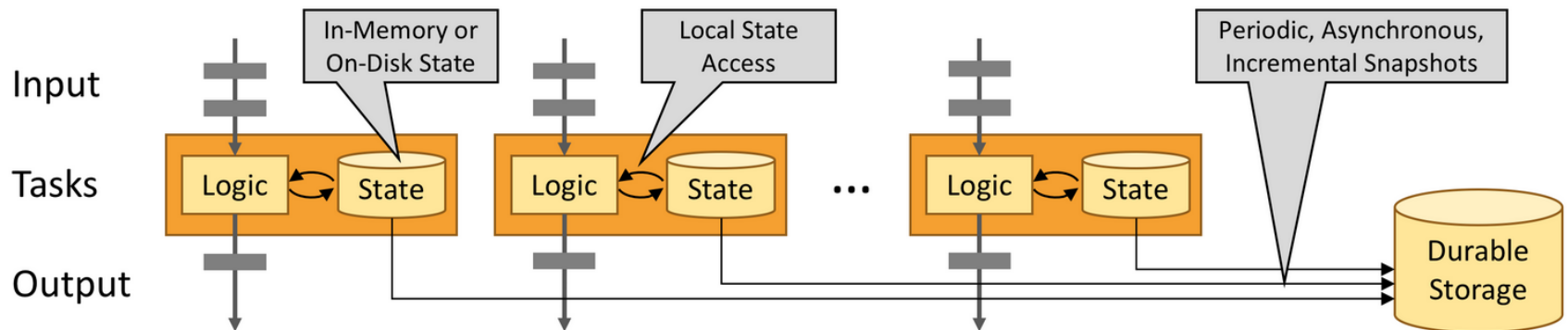
Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over *unbounded and bounded* data streams. Flink has been designed to run in *all common cluster environments*, perform computations at *in-memory speed* and at *any scale*



Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.

Flink Architecture



Flink Scalability

- applications processing **multiple trillions of events per day**,
- applications maintaining **multiple terabytes of state**, and
- applications **running on thousands of cores**.

END