Session 7

Data Stream Processing & Complex Event Processing Fundamentals

Big Data Analytics Technology, MSc in Data Science,
Coventry University UK

Miyuru Dayarathna

Presentation Outline

- Event Processing
- Complex event processing
- Stream Processing
- Conclusion

Event Processing (EP)

- Event processing is a technology which emerged in late 1990s
- For more than seven decades event processing has been technical basis for,
  - Discrete event simulation
  - Weather simulation and forecasting
  - Networks and the Internet
  - All manners of information gathering and communications

Event

- An event is simply a real world incident which has been captured by a system
- An event is an occurrence within a particular system or domain
  - It is something that has happened, or is contemplated as having happened in that domain
- The word event is also used to mean a programming entity that represents such an occurrence in a computing system

Event

- Contemplated as having happened ...
  - It is possible to have events that don’t correspond to actual occurrences.
  - Imagine a fraud detection system being used in a financial institution.
    - A system monitors financial transactions and generates events when it suspects Event-driven behavior and event-driven computing that a fraud is being conducted
  - These systems can generate false positives, so further investigation is usually required before you can be sure whether a fraud has actually taken place or not.

Event (Contd.)

- Basic Events
  - phone rings, an email arrives, somebody knocks at the door, or a book falls on the floor
- Unexpected events
  - Robbery event
  - late flight causing you to miss a connection
  - winning a lottery
  - getting a large order from an unexpected customer

Event (Contd.)

- Events that are observed easily
  - Things that we see and hear during our daily activities
- Need to do extra work to detect the event occurrence. What we observe are its symptoms
  - You recently getting noticed that your family's milk consumption had increased, and you need to add an additional carton of milk to your weekly grocery list.
  - You could arrive at this conclusion when you run out of milk for three consecutive weeks.

Event (Contd.)

- The main reason for learning that events have occurred is that it gives us the opportunity to react to them
- In the above example you might react to the *milk consumption has increased* event by increasing your weekly purchase of milk.
- Many of the events around us are outside the scope of our interest. Some events are background noise and do not require any reaction, but some do require reaction, and those we call *situations*.

Situation

- A situation is an event occurrence which might require a reaction
- One of the main themes in event processing is the detection and reporting of situations so that they can be reacted to.
- The reaction can be simple one or more complicated.
  - If we miss a flight connection there may be several alternative reactions depending on the time of the day, the airport where we are stranded, the airline policies, and the number of other passengers in the same situation.

Event Processing (EP)

- Event Processing (EP) is the computing which captures and processes the occurrence of real world incidents (events)
- Common event processing operations include reading, creating, transforming, and deleting events.
- The event producers can be of various different types including financial feeds, news feeds, weather sensors, application logs, video streams collected from surveillance cameras, etc.
- The EP engine is the brain of the processing which does multiple types of processing on event streams based on predefined rules.

Event Processing vs Event-based Programming

- **Abstraction** - Operations that form the event processing logic can be separated from the application logic, allowing them to be modified without having to change the producing and consuming applications

- **Decoupling** - The events detected and produced by one particular application can be consumed and acted on by completely different applications. There's no need for producing and consuming applications to be aware of each others' existence, and they may be distributed anywhere in the world.

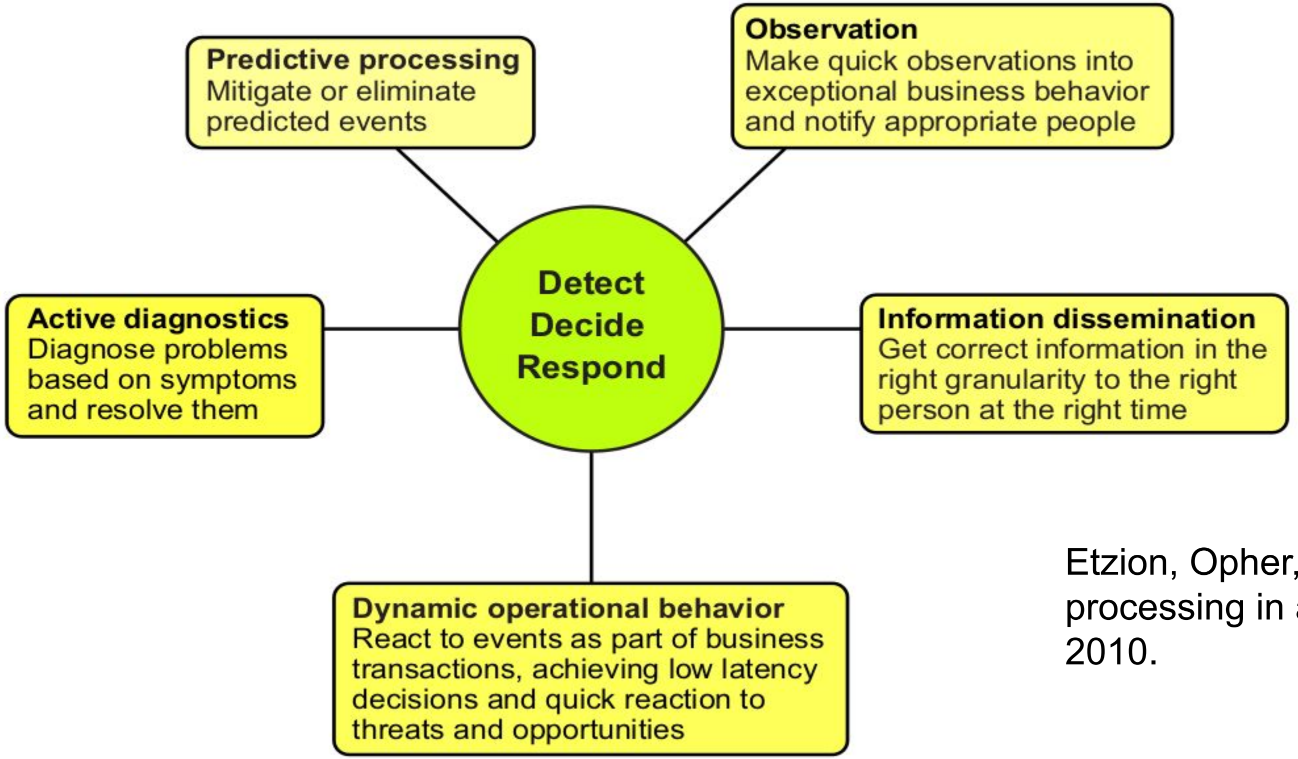
- **Real World Focus** - Event processing frequently deals with events that occur, or could occur, in the real world.

Event Processing and its relationship to the real world

- **Deterministic** - There is an exact mapping between a situation in the real world and its representation in the event processing system.

- **Approximate** - The event processing system provides an approximation to real world events.

Categories of event processing applications


Predictive processing
Mitigate or eliminate predicted events
Observation
Make quick observations into exceptional business behavior and notify appropriate people
Detect
Decide
Respond
Active diagnostics
Diagnose problems based on symptoms and resolve them
Information dissemination
Get correct information in the right granularity to the right person at the right time
Dynamic operational behavior
React to events as part of business transactions, achieving low latency decisions and quick reaction to threats and opportunities

Etzion, Opher, and Peter Niblett. Event processing in action. Manning Publications Co., 2010.

Categories of event processing applications - Observation

- Observation - Event processing is used to monitor a system or process by looking for exceptional behavior and generating alerts when such behavior occurs.
- Event processing is used to monitor a system or process by looking for exceptional behavior and generating alerts when such behavior occurs.
- E.g.,
  - Patient monitoring system
  - Luggage monitoring system
  - Safety regulation compliance system

Categories of event processing applications - Information Dissemination

- Another reason for using event processing is to deliver the right information to the right consumer at the right granularity at the right time, in other words, personalized information delivery.
- E.g.,
    - Personalized Banking System
    - Emergency system that sends alerts to first responders

Categories of event processing applications - Dynamic operational behavior

- Event processing is often used to drive the actions performed by a system dynamically so as to react to incoming events.
- The output of the application is directly affected by the input events
- E.g.,
  - Online trading system

Categories of event processing applications - Active Diagnostics

- Here the goal of the event processing application is to diagnose a problem, based on observed symptoms.
- E.g.,
  - Mechanical failure
  - Help-desk system

Categories of event processing applications - Predictive processing

- Here the goal is to identify events before they have happened, so that they can be eliminated or their effects mitigated.
- E.g.,
  - Fraud detection system

Reasons for using event processing

- Application might be naturally centered on events, involve sensors that detect and report events, and the purpose of the application is to analyze and react to these events.
- Application might need to identify and react to certain situations (either good or bad) as they occur.
  - An event-driven approach, where changes in state are monitored as they happen, lets an application respond in a much more timely fashion than a batch approach where the detection process runs only intermittently.

Reasons for using event processing (Contd.)

- Event processing can give you a way of extending an existing application in a flexible, non-invasive manner.
- Rather than changing the original application to add the extra function, it’s sometimes possible to instrument the original application by adding event producers to it (for example, by processing the log files that it produces).
- The additional functionality can then be implemented by processing the events generated by these event producers.

Reasons for using event processing (Contd.)

- Intermediary event processing logic can be separated out from the rest of the application. This can allow the application to be adapted quickly to meet new business requirements, sometimes by the application business users themselves.
- The application might involve analysis of a large amount of data in order to provide an output to be delivered to a human user or another application. This data can be organized into streams of events which are then distributed to multiple computing nodes allowing separate parts of the analysis to be performed in parallel.

Reasons for using event processing (Contd.)

- There are potential scalability and fault tolerance benefits to be gained by using an event-driven approach.
- An event-driven approach allows processing to be performed asynchronously, and so is well suited to applications where events happen in an irregular manner.
- If event activity suddenly spikes, it may be possible to defer some processing to a subsequent, quieter time.

Complex Event Processing (CEP)

- A complex event is an event derived from a group of events using either aggregation of derivation functions
- Information enclosed in a set of related events can be represented (i.e., summarized) through such a complex event.
- CEP can be defined as computing which conducts operations on complex events.
- Complex Event Processing (CEP) is a defined set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems

Luckham, David (2008), The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, RuleML 2008

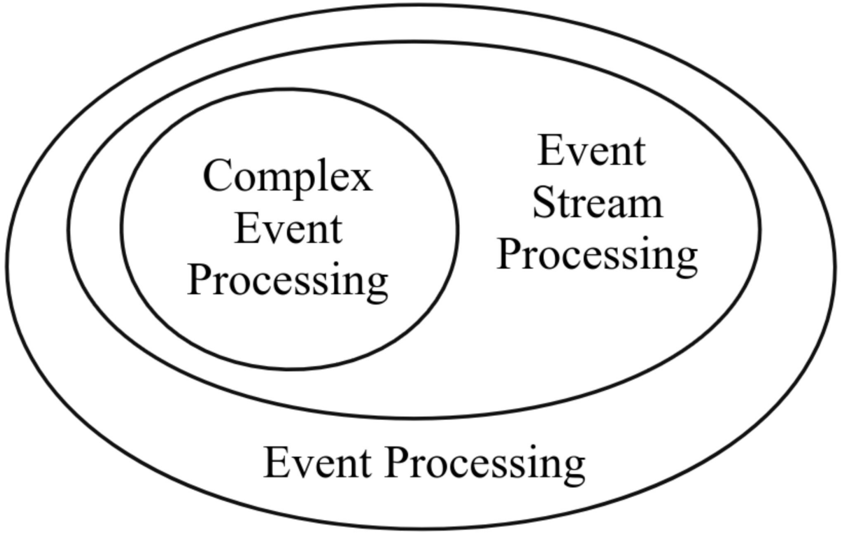
Complex Event Processing (CEP)

- CEP is the computing which conducts operations on complex events.
- CEP includes advanced processing such as pattern matching, event prediction, etc.
- After year 2000, CEP began to appear in commercial IT applications

Complex Event Processing (CEP) (Contd.)

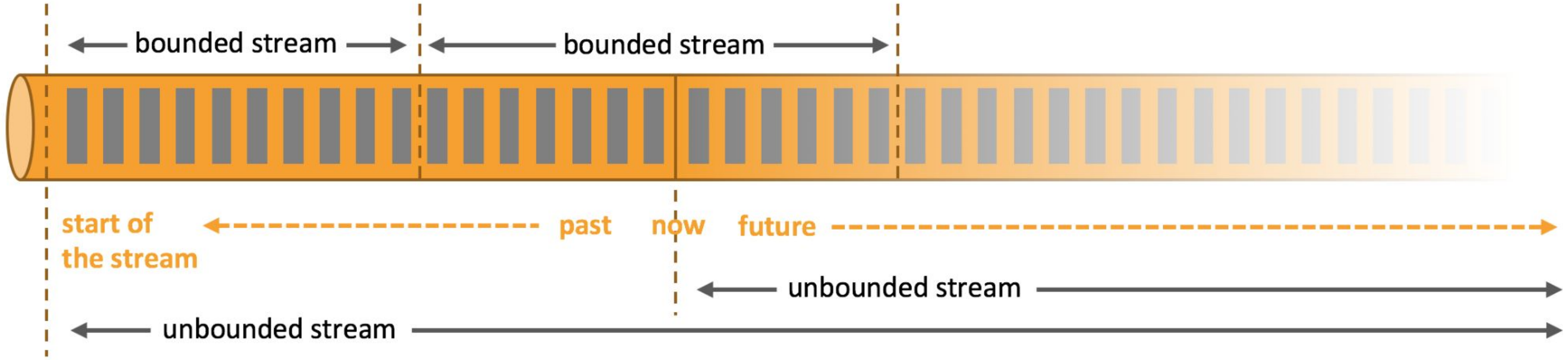
Event - a real world incident which has been captured by a system.

Complex event - an event derived from a group of events using either aggregation of derivation functions



Data Processing Patterns

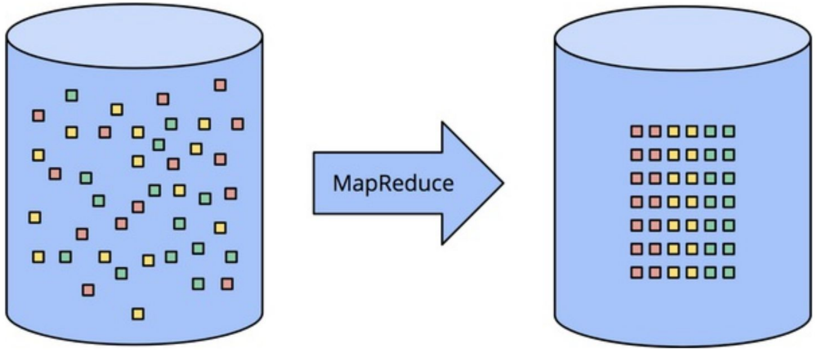
- Bounded Data
- Unbounded Data : Batch
- Unbounded Data : Stream



https://nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/overview/

Data Processing Patterns - Bounded Data

- Processing bounded data is conceptually very straight forward
- We start out on the left with a dataset full of entropy.
- 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 side end up with a new structured dataset with greater inherent value.



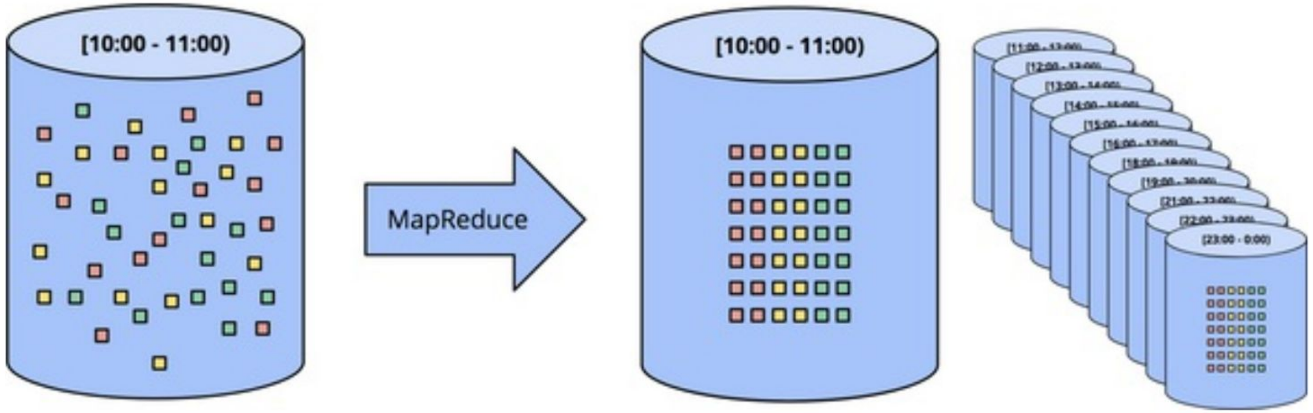
Bounded data processing with a classic batch engine. A finite pool of unstructured data on the left is run through a data processing engine, resulting in corresponding structured data on the right.

Unbounded Data : Batch

- Batch engines, though not explicitly designed with unbounded data in mind, have nevertheless been used to process unbounded datasets since batch systems were first conceived.
- Such approaches revolve around slicing up the unbounded data into a collection of bounded datasets appropriate for batch processing.

Unbounded Data : Batch - Fixed Windows

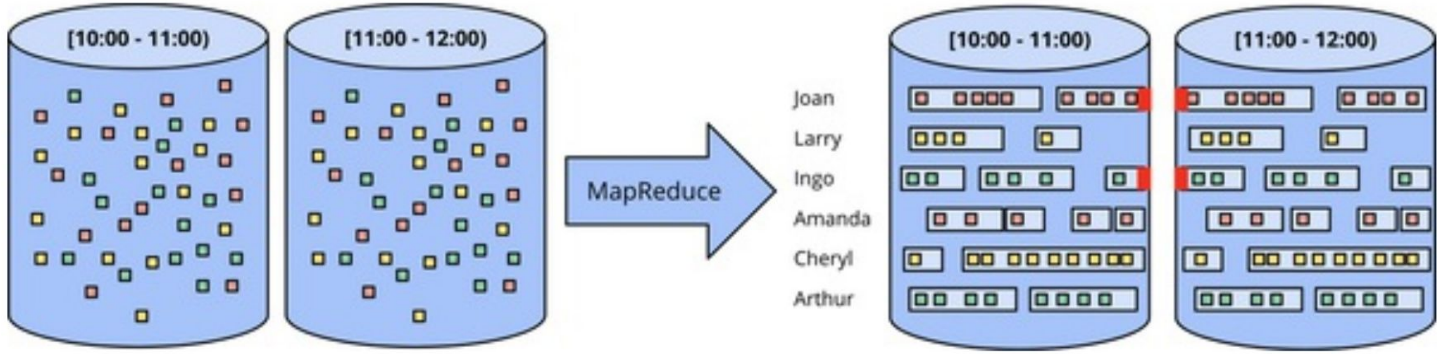
- The most common way to process an unbounded dataset using repeated runs of a batch engine is by windowing the input data into fixed-size windows and then processing each of those windows as a separate, bounded data source (sometimes also called tumbling windows)



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

Unbounded Data : Batch - Sessions

- Batch engine can be used to process unbounded data into more sophisticated windowing strategies, like sessions.
- Sessions are typically defined as periods of activity (e.g., for a specific user) terminated by a gap of inactivity. When calculating sessions using a typical batch engine, you often end up with sessions that are split across batches, as indicated by the red marks in the below figure.



Unbounded data processing into sessions via ad hoc fixed windows with a classic batch engine. An unbounded dataset 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 : Batch - Sessions

- We can reduce the number of splits by increasing batch sizes, but at the cost of increased latency.
- Another option is to add additional logic to stitch up sessions from previous runs, but at the cost of further complexity.
- Either way, using a classic batch engine to calculate sessions is less than ideal.
- Better approach is to build up sessions in a streaming manner

Unbounded Data : Streaming

- Different from the ad hoc nature of most batch-based unbounded data processing approaches, streaming systems are built for unbounded data.
- For many real-world, distributed input sources, we may not only find ourselves dealing with unbounded data, but also deal with data having the following characteristics,
  - Highly unordered with respect to event times
    - We need some sort of time-based shuffle in our pipeline if we want to analyze the data in the context in which they occurred
  - Of varying event-time skew
    - We can’t just assume we’ll always see most of the data for a given event time X within some constant epsilon of time Y.

Approaches for Handling unbounded streams of data

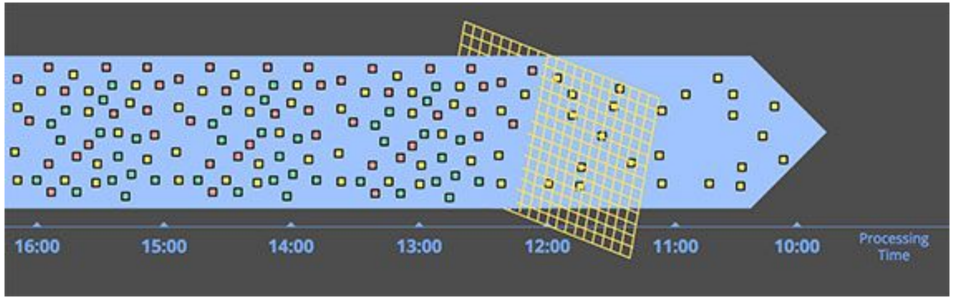
- Time-agnostic
- Filtering
- Inner joins
- Approximation
- Windowing
  - Fixed Windows
  - Sliding Windows
  - Sessions

Time-agnostic processing

- Time-agnostic processing is used for cases in which time is essentially irrelevant; that is, all relevant logic is data driven.
- Because 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.
- As a result, essentially all streaming systems in existence support time-agnostic use cases out of the box.
- Batch systems are also well suited for time-agnostic processing of unbounded data sources by simply chopping the unbounded source into an arbitrary sequence of bounded datasets and processing those datasets independently.

Filtering

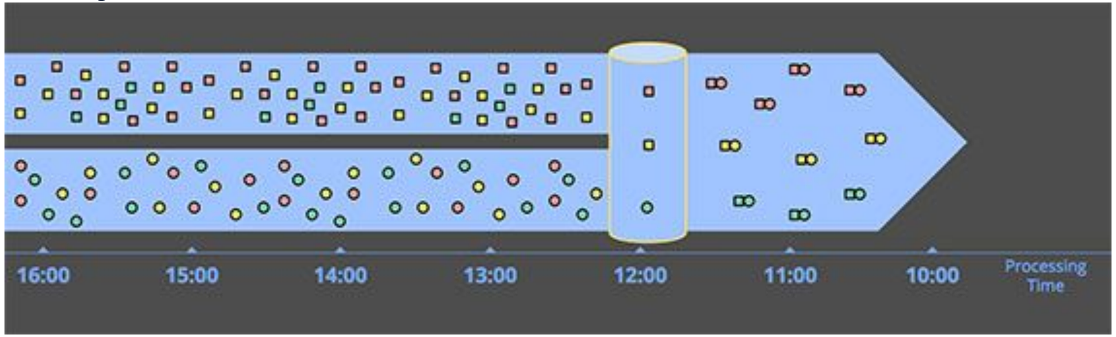
- A very basic form of time-agnostic processing is filtering
- Imagine you’re processing Web traffic logs, and you want to filter out all traffic that didn’t originate from a specific domain.
- You would look at each record as it arrived, see if it belonged to the domain of interest, and drop it if not.
- Since this sort of thing depends only on a single element at any time, the fact that the data source is unbounded, unordered, and of varying event time skew is irrelevant.



A collection of data (flowing left to right) of varying types is filtered into a homogeneous collection containing a single type.

Inner Joins

- Inner join is another example for time-agnostic processing.
- When joining two unbounded data sources, if you care only about the results of a join when an element from both sources arrive, there's no temporal element to the logic.
- Upon seeing a value from one source, you can simply buffer it up in persistent state; only after the second value from the other source arrives do you need to emit the joined record.



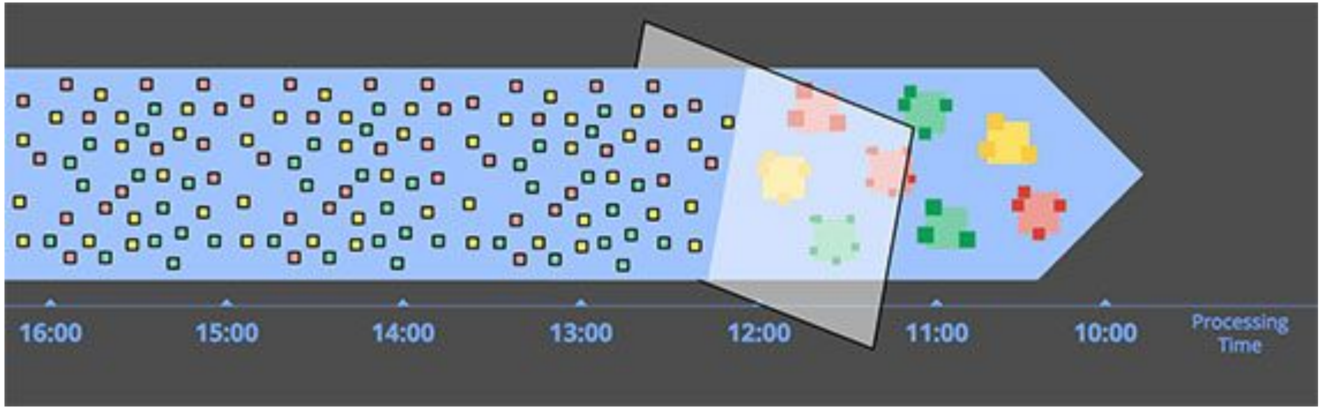
Joins are produced when matching elements from both sources are observed

Inner Joins (Contd.)

- Switching semantics to some sort of outer join introduces the data completeness problem we've talked about: after you've seen one side of the join, how do you know whether the other side is ever going to arrive or not?
  - We need to introduce some notion of a timeout, which introduces an element of time. That element of time is essentially a form of windowing.

Approximation Algorithms

- The second major category of approaches is approximation algorithms, such as approximate Top-N, streaming k-means, and so on.
- They take an unbounded source of input and provide output data that, a visualization of the approximation can be listed as follows,



Data are run through a complex algorithm, yielding output data that look more or less like the desired result on the other side.

Approximation Algorithms (Contd.)

- The upside of approximation algorithms is that, by design, they are low overhead and designed for unbounded data.

- The downsides are that a limited set of them exist, the algorithms themselves are often complicated, and their approximate nature limits their utility.
- These algorithms typically do have some element of time in their design (e.g., some sort of built-in decay) because they process elements as they arrive, that time element is usually processing-time based.

Approximation Algorithms (Contd.)

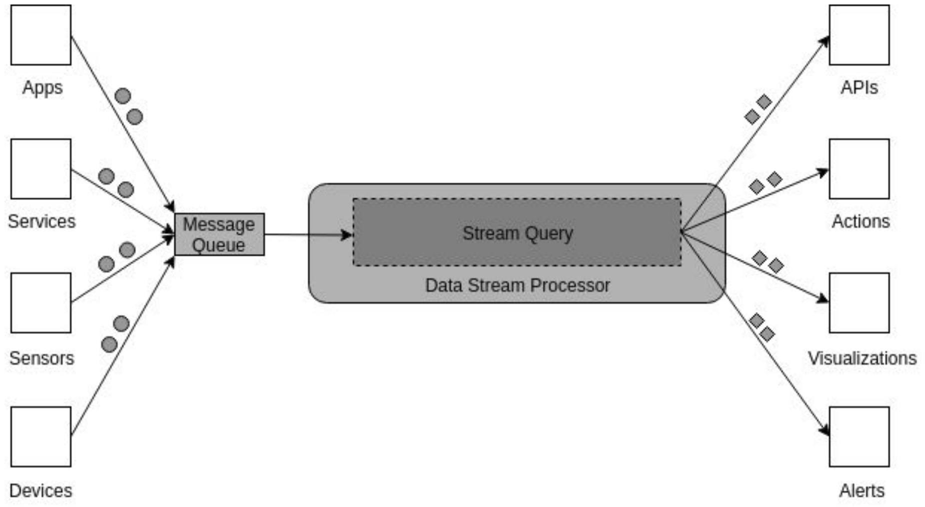
- This is particularly important for algorithms that provide some sort of provable error bounds on their approximations.

- If those error bounds are predicated on data arriving in order, they mean essentially nothing when you feed the algorithm unordered data with varying event-time skew.

- Approximation algorithms themselves are a fascinating subject, but as they are essentially another example of time-agnostic processing (modulo the temporal features of the algorithms themselves), they’re quite straightforward to use

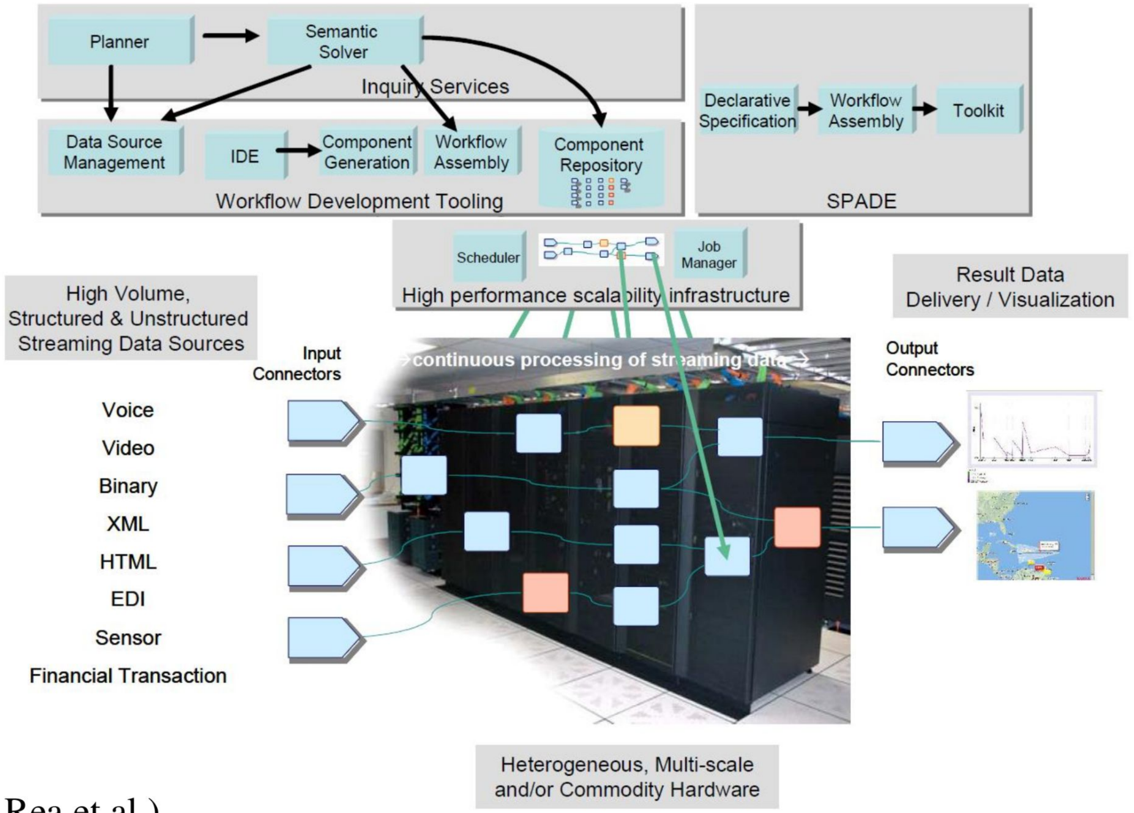
Stream Processing

- Stream processing is a novel data processing paradigm that refers to the processing of data in motion, or in other words, computing on data directly as it is produced or received.



https://www.ververica.com/what-is-stream-processing

Data Stream Processing Application Deployed in a compute cluster


Planner
Semantic Solver
Inquiry Services
Data Source Management
IDE
Component Generation
Workflow Assembly
Component Repository
Workflow Development Tooling
Declarative Specification
Workflow Assembly
Toolkit
SPADE
Scheduler
Job Manager
High performance scalability infrastructure
Result Data Delivery / Visualization
High Volume, Structured & Unstructured Streaming Data Sources
Input Connectors
→continuous processing of streaming data→
Output Connectors
Voice
Video
Binary
XML
HTML
EDI
Sensor
Financial Transaction
Heterogeneous, Multi-scale and/or Commodity Hardware

(Roger Rea et al.)

Stream Processing (Contd.)

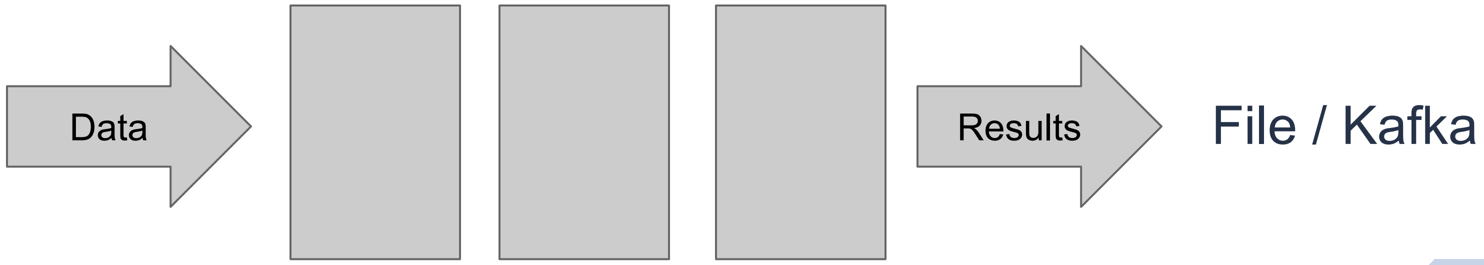
- Some data naturally comes as a never-ending stream of events.
  - The input data is unbounded - A series of events, no predetermined beginning or ending
  - Credit card transactions, clicks on a website, or sensor readings from IoT devices.
- Sometimes the amount of data is huge and it is not even possible to store it.
- Time value of data.
- Stream processing is the act of continuously incorporating new data to compute a result

Streaming Data

- The stream of data is split into a sequence of individual tuples
- A data tuple is the atomic data item in a data stream
- A data tuple can be structured, semi-structured, and unstructured

Stream Processing Patterns

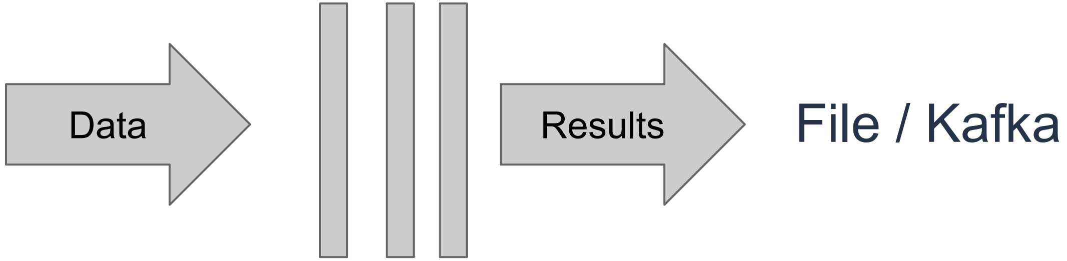
- Micro-batch
  - Batch engines
  - Slice up the unbounded data into a set of bounded data and then process each batch



Microbatches of DataFrames

Stream Processing Patterns (Contd.)

- Continuous processing
  - Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes

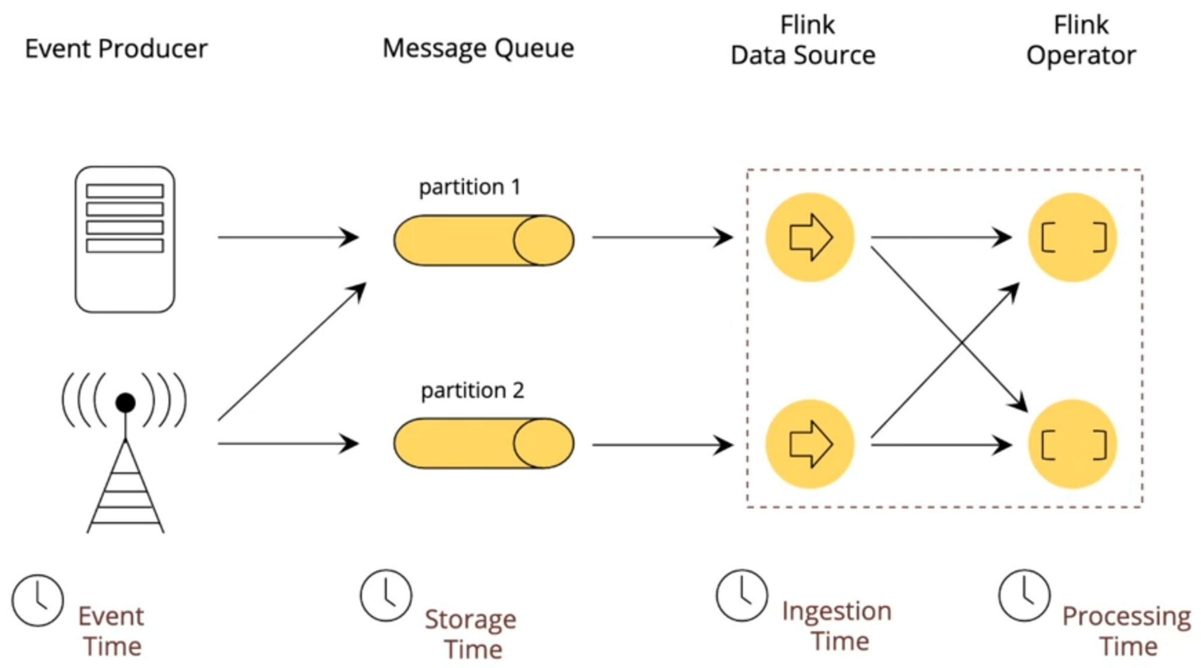


One record at a time

Notions of Time

- **Event time** : The time at which the events actually occurred
    - Timestamps inserted into each record at the source
- **Storage time** : Timestamp that might be injected by the storage layer
- **Ingestion time :** The point in time when an event or data record is ingested into the stream processor
- **Processing time** : The point in time when the event or data record happens to be processed by the stream processing application, i.e. when the record is being consumed.

Notions of Time



https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/time/

Event Time

- Event time is the time that each individual event occurred on its producing device.
- This time is typically embedded within the records before they enter the stream processor, and that event timestamp can be extracted from each record.
- In event time, the progress of time depends on the data, not on any wall clocks.
  - Event time can progress independently of processing time (measured by wall clocks).
  - E.g., In one program the current event time of an operator may trail slightly behind the processing time (accounting for a delay in receiving the events), while both proceed at the same speed.
  - On the other hand, another streaming program might progress through weeks of event time with only a few seconds of processing, by fast-forwarding through some historic data already buffered in a Kafka topic (or another message queue).
  - Need to deal with out of order events when using event time

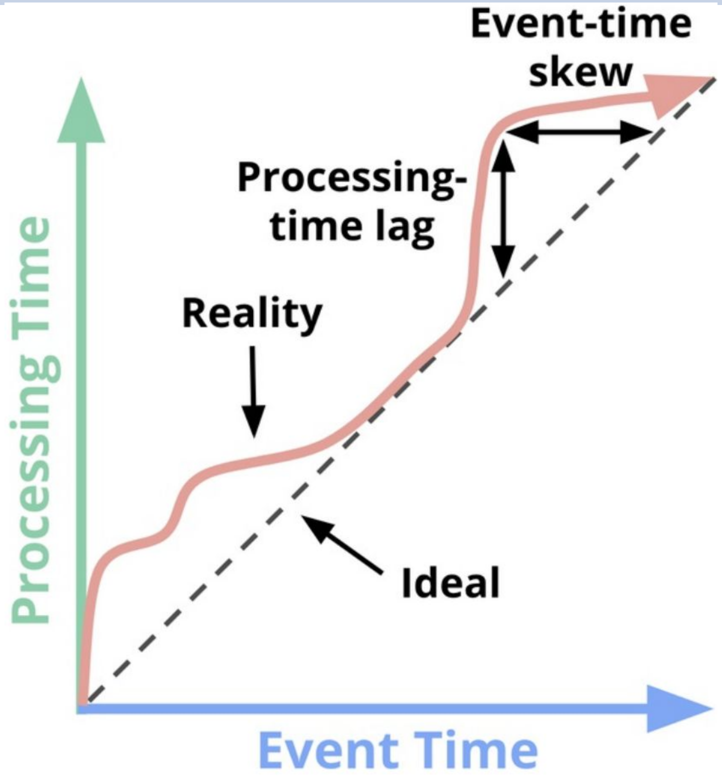
Event Time (Contd.)

- In a perfect world, event time processing would yield completely consistent and deterministic results, regardless of when events arrive, or their ordering.
- However, unless the events are known to arrive in-order (by timestamp), event time processing incurs some latency while waiting for out-of-order events.
- As it is only possible to wait for a finite period of time, this places a limit on how deterministic event time applications can be.
- Assuming all of the data has arrived, event time operations will behave as expected, and produce correct and consistent results even when working with out-of-order or late events, or when reprocessing historic data.
- Note that sometimes when event time programs are processing live data in real-time, they will use some processing time operations in order to guarantee that they are progressing in a timely fashion.

Processing Time

- Processing time refers to the system time of the machine that is executing the respective operation.
- Processing time for an event is the system clock time
- Event time is deterministic and immutable, the processing time is always changing
- If we run event time based pipeline on the same data you get the same results. But if you run the processing time based pipeline more than once on the same data you may or may not get the same results. Determinism and the ability to reprocess historic data are advantages of event time.
- Nothing can go out of order with respect to time when using processing time.

Time Domains (Contd.)



- Ideally event time and processing time needs to be equal
- However, there is a skew between event time and processing time

https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

Windowing

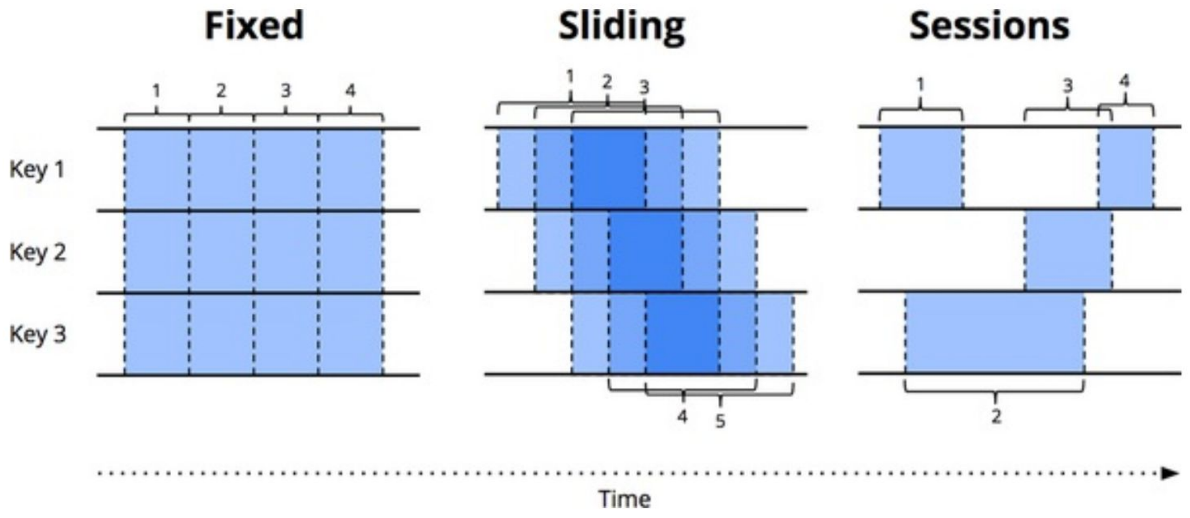
- Window is a buffer associated with a stream to retain previously received tuples

- Different window management policies
  - **Count-based policy** : the maximum number of tuples a window buffer can hold

  - **Time-based policy** : based on processing or event time period

Windowing (Contd.)

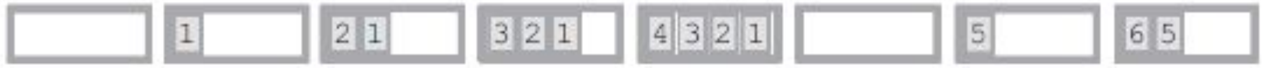
- Windowing is simply the notion of taking a data source (either unbounded or bounded) and chopping it up along temporal boundaries into finite chunks for processing



Each example is shown for three different keys, highlighting the difference between aligned windows (which apply across all the data) and unaligned windows (which apply across a subset of the data).

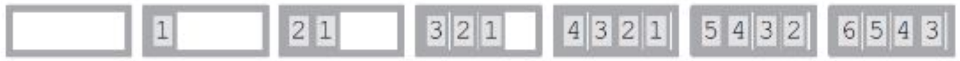
Window Types

- **Fixed/Tumbling window** : supports batch operations
  - When the buffer fills up, all the tuples are evicted
  - Fixed windows slice up time into segments with a fixed-size temporal length.
  - Typically, the segments for fixed windows are applied uniformly across the entire data set, which is an example of aligned windows.
  - In some cases, it’s desirable to phase-shift the windows for different subsets of the data (e.g., per key) to spread window completion load more evenly over time, which instead is an example of unaligned windows since they vary across the data.



Window Types (Contd.)

- **Sliding Window** : supports incremental operations
  - When the buffer fills up, older tuples are evicted
  - A generalization of fixed windows, sliding windows are defined by a fixed length and a fixed period.
  - If the period is less than the length, then the windows overlap. If the period equals the length, we have fixed windows.
  - If the period is greater than the length, we have a weird sort of sampling window that only looks at subsets of the data over time. As with fixed windows, sliding windows are typically aligned, though may be unaligned as a performance optimization in certain use cases.



Window Types (Contd.)

- **Sessions** : An example of dynamic windows, sessions are composed of sequences of events terminated by a gap of inactivity greater than some timeout.
  - Sessions are commonly used for analyzing user behavior over time, by grouping together a series of temporally-related events (e.g., a sequence of videos viewed in one sitting).
  - Sessions are interesting because their lengths cannot be defined a priori; they are dependent upon the actual data involved.
  - They're also the canonical example of unaligned windows since sessions are practically never identical across different subsets of data (e.g., different users).

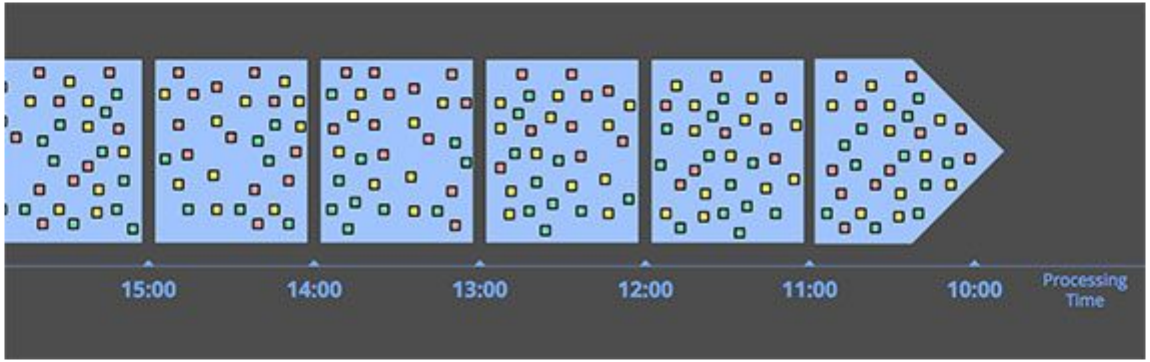
Triggering vs Windowing

- Triggering determines when in processing time the results of groupings are emitted
  - Time-based or data-driven triggers

- Windowing determines where in event time data are grouped together for processing
  - Time-based or data-driven windows

Windowing by processing time

- The system buffers up incoming data into windows until some amount of processing time has passed.



Data gets collected into windows based on the order they arrive in the pipeline

https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

Windowing by processing time (Contd.)

- In the case of five-minute fixed windows, the system would buffer up data for five minutes of processing time, after which it would treat all the data it had observed in those five minutes as a window and send them down stream for processing

Windowing by processing time (Contd.)

- Advantages of processing time windowing,
  - Simple :
    - The implementation is extremely straightforward since you never worry about shuffling data within time.
    - You just buffer things up as they arrive and send them downstream when the window closes.
  - Judging window completeness is straightforward :
    - Since the system has perfect knowledge of whether all inputs for a window have been seen or not, it can make perfect decisions about whether a given window is complete or not.
    - This means there is no need to be able to deal with “late” data in any way when windowing by processing time.

Windowing by processing time (Contd.)

- Advantages of processing time windowing,
  - Infer information about the source as it is observed
    - Processing time windowing exactly matches this scenario.

    - Many monitoring scenarios fall into this category. Imagine tracking the number of requests per second sent to a global-scale web service.

    - Calculating a rate of these requests for the purpose of detecting outages is a perfect use of processing time windowing.

Windowing by processing time (Contd.)

- Disadvantages
  - If the data in question have event times associated with them, those data must arrive in event time order if the processing time windows are to reflect the reality of when those events actually happened
  - However, event-time ordered data are uncommon in many real-world, distributed input sources.

Windowing by processing time (Contd.)

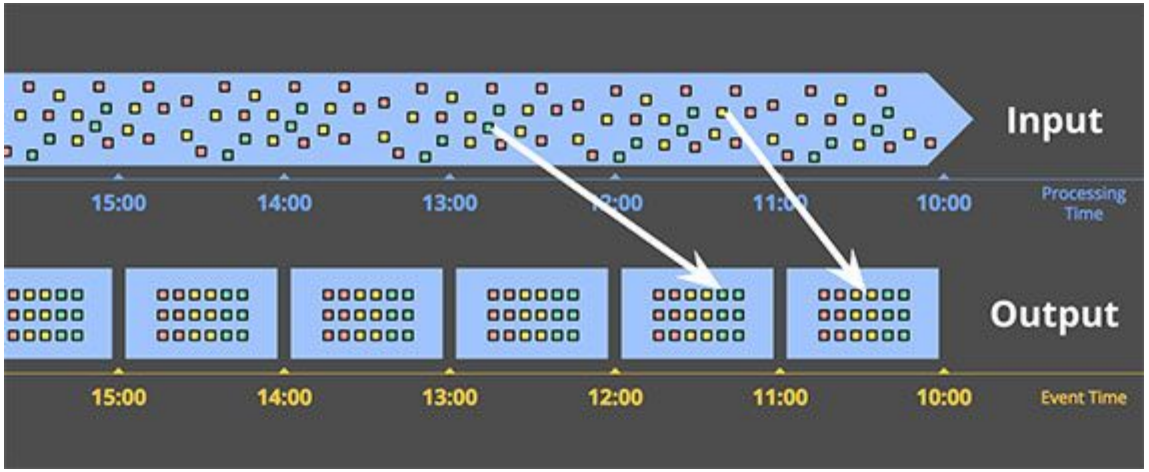
- Any mobile app that gathers usage statistics for later processing.
- In cases where a given mobile device goes offline for any amount of time (brief loss of connectivity, airplane mode while flying across the country, etc.), the data recorded during that period won’t be uploaded until the device comes online again.
- That means data might arrive with an event time skew of minutes, hours, days, weeks, or more. It’s essentially impossible to draw any sort of useful inferences from such a data set when windowed by processing time.

Windowing by processing time (Contd.)

- Many distributed input sources may seem to provide event-time ordered (or very nearly so) data when the overall system is healthy
  - event-time skew is low for the input source when healthy, will not be the same
  - E.g., Global service which processes data collected in multiple continents. Network issues in a bandwidth-constrained transcontinental line further decrease bandwidth and/or increase latency, suddenly a portion of your input data may start arriving with much greater skew than before.
  - If you are windowing that data by processing time, your windows are no longer representative of the data that actually occurred within them; instead, they represent the windows of time as the events arrived at the processing pipeline, which is some arbitrary mix of old and current data.

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 is the gold standard of windowing



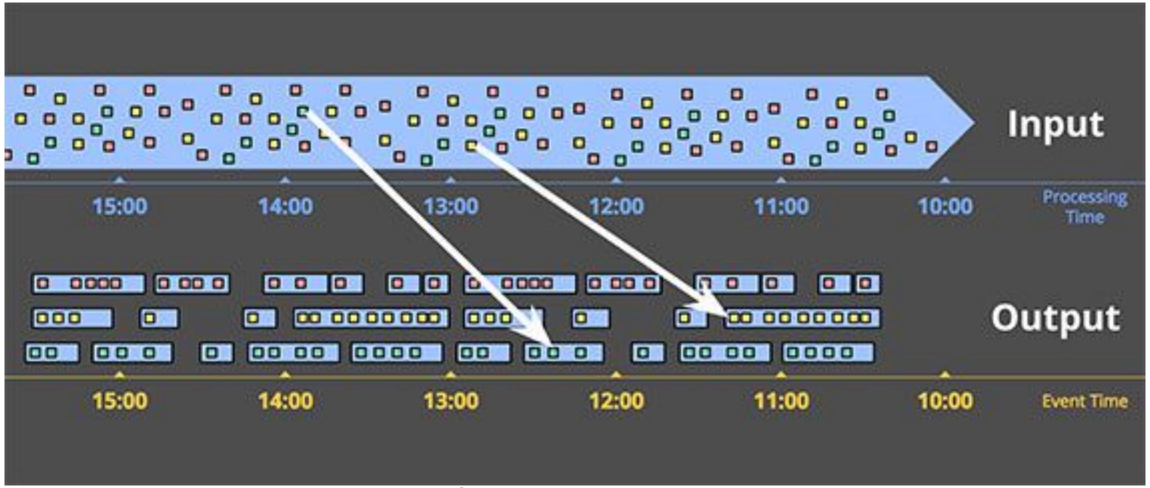
Data are collected into windows based on the times they occurred. The white arrows call out example data that arrived in processing time windows that differed from the event time windows to which they belonged.

Windowing by Event Time - Advantages

- Event time correctness
  - The solid white lines in the diagram call out two particular data of interest.
  - Those two data both arrived in processing time windows that did not match the event time windows to which they belonged.
  - As such, if these data had been windowed into processing time windows for a use case that cared about event times, the calculated results would have been incorrect.

Windowing by Event Time - Advantages

- Create dynamically sized windows (i.e., sessions) without the arbitrary splits observed when generating sessions over fixed windows



Data are collected into session windows capturing bursts of activity based on the times that the corresponding events occurred. The white arrows again call out the temporal shuffle necessary to put the data into their correct event-time locations.

Windowing by Event Time - Disadvantages

- Due to windows must often live longer (in processing time) than the actual length of the window itself
- **Buffering**
  - Due to extended window lifetimes, more buffering of data is required. Thankfully, persistent storage is generally the cheapest of the resource types most data processing systems depend on (the others being primarily CPU, network bandwidth, and RAM).
  - As such, this problem is typically much less of a concern than one might think when using any well-designed data-processing system with strongly consistent persistent state and a decent in-memory caching layer.
  - Many useful aggregations do not require the entire input set to be buffered (e.g., sum, or average), but instead can be performed incrementally, with a much smaller, intermediate aggregate stored in persistent state.

Windowing by Event Time - Disadvantages (Contd.)

- **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. For many types of inputs, the system can give a reasonably accurate heuristic estimate of window completion
  - In cases where absolute correctness is paramount (e.g., billing), the only real option is to provide a way for the streaming app developer to express when they want results for windows to be materialized, and how those results should be refined over time.

Trigger

- 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.

- We can consider them as a flow control mechanism for dictating when results should be materialized. Another way of looking at it is that triggers are like the shutter-release on a camera, allowing you to declare when to take a snapshots in time of the results being computed.

Trigger (Contd.)

- Triggers also make it possible to observe the output for a window multiple times as it evolves.
- This in turn opens up the door to refining results over time, which allows for providing speculative results as data arrive, as well as dealing with changes in upstream data (revisions) over time or data that arrive late (e.g., mobile scenarios, in which someone’s phone records various actions and their event times while the person is offline and then proceeds to upload those events for processing upon regaining connectivity).

Watermark

- A watermark is a notion of input completeness with respect to event times.
- A watermark with value of time X makes the statement: "*all input data with event times less than X have been observed.*"
- As such, watermarks act as a metric of progress when observing an unbounded data source with no known end.
- The watermark is a monotonically increasing timestamp of the oldest work not yet completed

Watermark (Contd.)

- Watermarks are a supporting aspect 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.

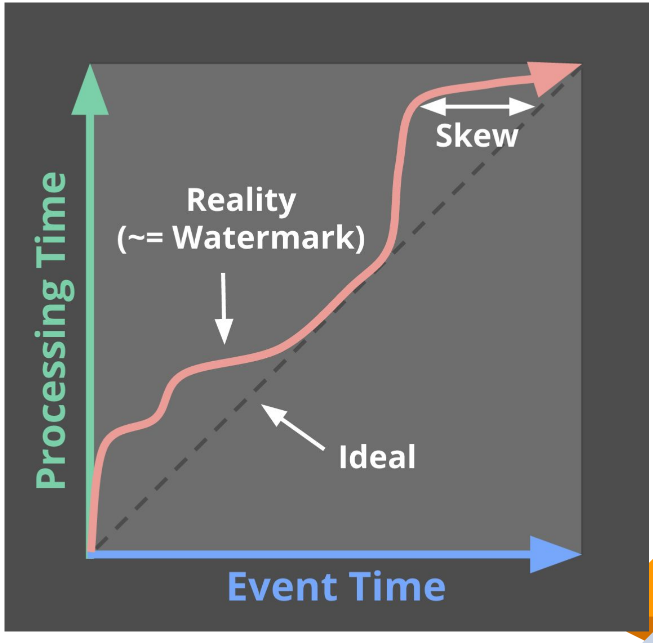
- They are the way the system measures progress and completeness relative to the event times of the records being processed in a stream of events (either bounded or unbounded, though their usefulness is more apparent in the unbounded case).

Watermark (Contd.)

- The skew between event time and processing time as an ever-changing function of time for most real-world distributed data processing systems

Red line represented reality is essentially the watermark

Red line captures the progress of event time completeness as processing time progresses



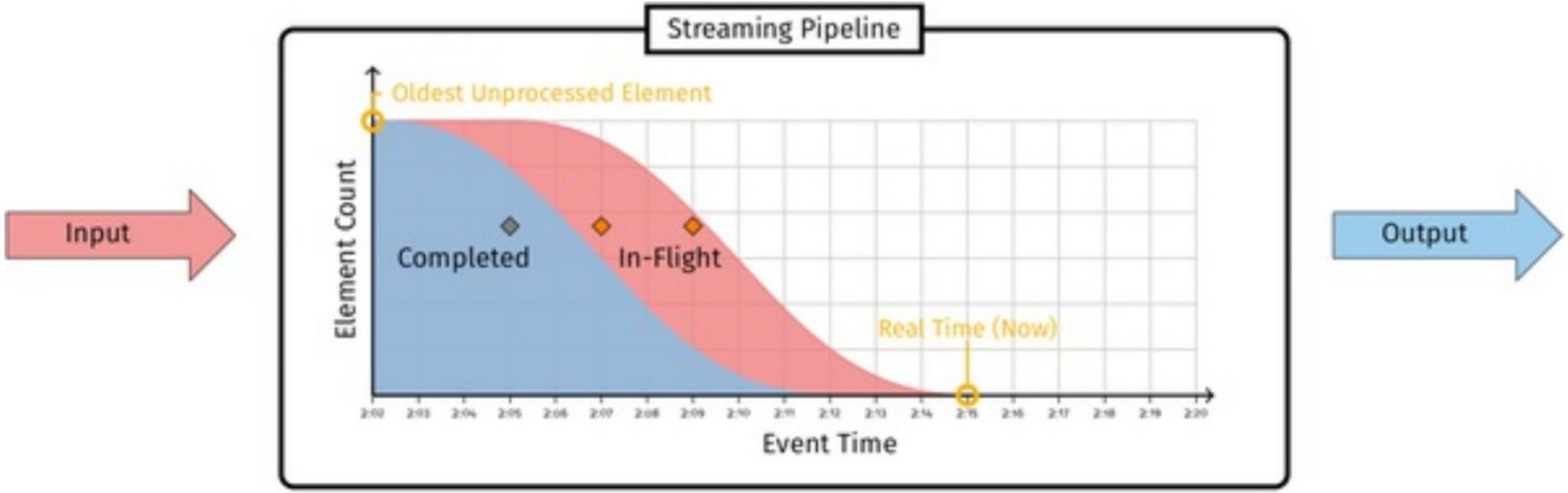
Watermark (Contd.)

- Conceptually, we can think of the watermark as a function, F(P) -> E, which takes a point in processing time and returns a point in event time
- The input to the function is really the current state of everything upstream of the point in the pipeline where the watermark is being observed: the input source, buffered data, data actively being processed, etc.
- Yet, conceptually, it’s simpler to think of it as a mapping from processing time to event time.
- 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.
- Depending upon the type of watermark, perfect or heuristic, that assertion may be a strict guarantee or an educated guess.

Watermark - Types

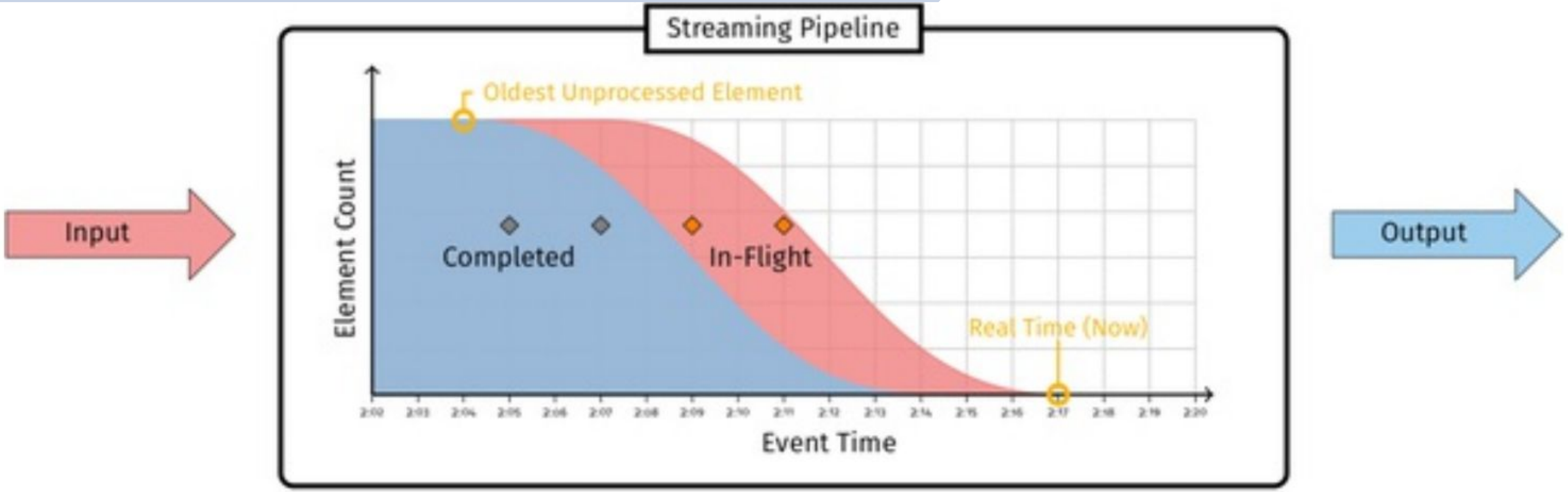
- **Perfect watermarks** : In the case where we have perfect knowledge of all of the input data, it’s possible to construct a perfect watermark; in such a case, there is no such thing as late data; all data are early or on time.
- **Heuristic watermarks** : For many distributed input sources, perfect knowledge of the input data is impractical, in which case the next best option is to provide a heuristic watermark.
  - Heuristic watermarks use whatever information is available about the inputs (partitions, ordering within partitions if any, growth rates of files, etc.) to provide an estimate of progress that is as accurate as possible.
  - In many cases, such watermarks can be remarkably accurate in their predictions. Even so, the use of a heuristic watermark means it may sometimes be wrong, which will lead to late data.

Distribution of in-flight and completed message event times within a streaming pipeline

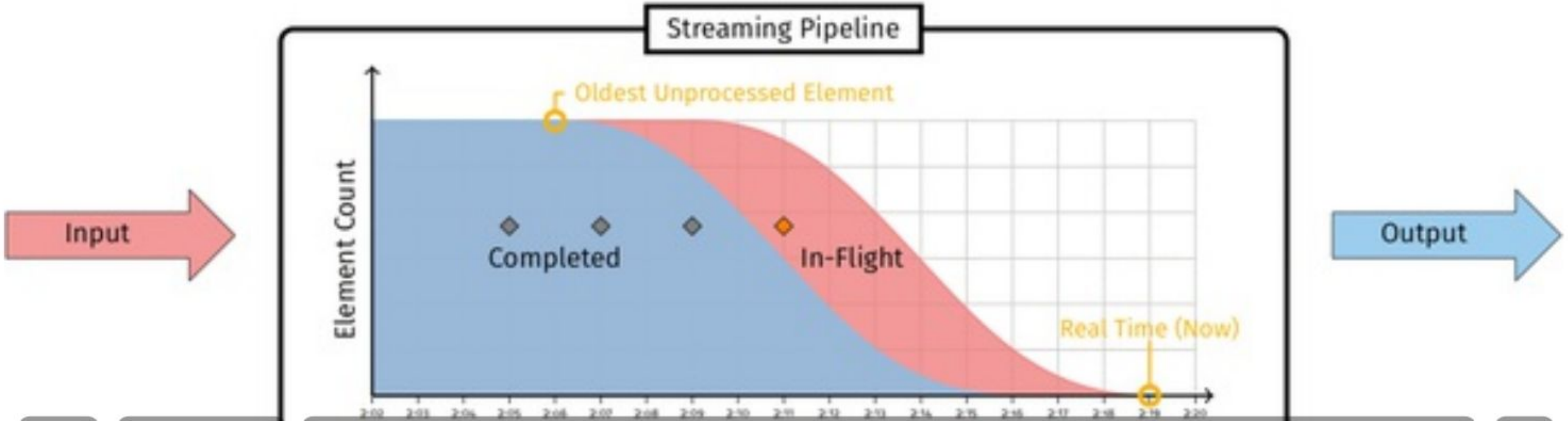


Messages are ingested by the pipeline, processed, and eventually marked completed. Each message is either “in-flight,” meaning that it has been received but not yet completed, or “completed,” meaning that no more processing on behalf of this message is required. If we examine the distribution of messages by event time, it will look something like above.

Distribution of in-flight and completed message event times within a streaming pipeline



Distribution of in-flight and completed message event times within a streaming pipeline



As time advances, more messages will be added to the “in-flight” distribution the right, and more of those messages from the “in-flight” part of the distribution will be completed and moved into the “completed” distribution.

Watermark (Contd.)

- The watermark is a monotonically increasing timestamp of the oldest work not yet completed.
- Completeness
  - If the watermark has advanced past some timestamp T, we are guaranteed by its monotonic property that no more processing will occur for on-time (non-late data) events at or before T. Therefore, we can correctly emit any aggregations at or before T. In other words, the watermark allows us to know when it is correct to close a window.

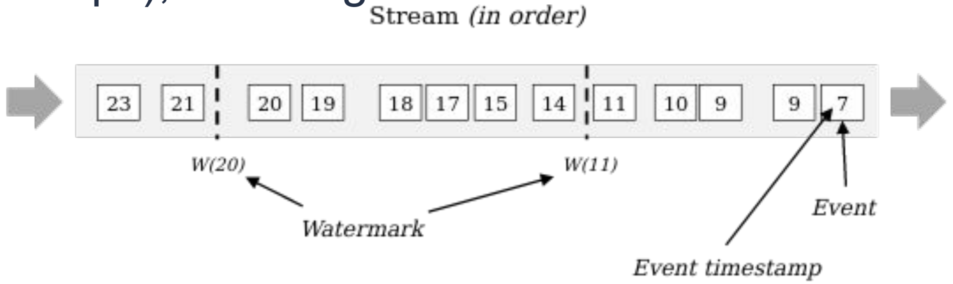
Watermark (Contd.)

- Visibility
  - If a message is stuck in our pipeline for any reason, the watermark cannot advance. Furthermore, we will be able to find the source of the problem by examining the message that is preventing the watermark from advancing.

Watermark (Contd.)

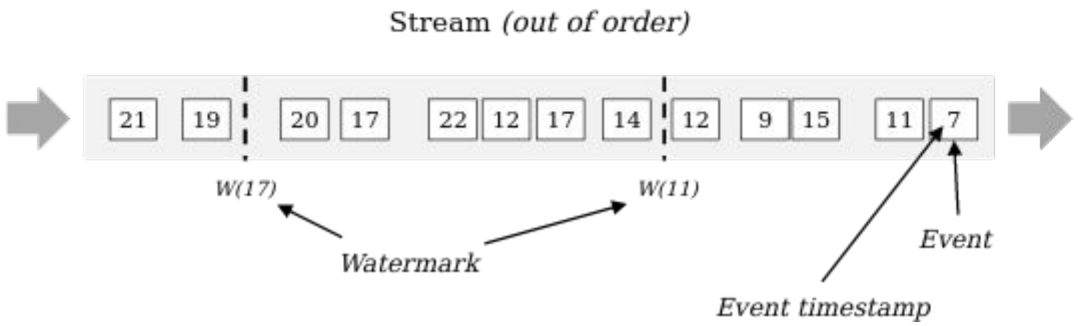
The mechanism in Flink to measure progress in event time is watermarks. Watermarks flow as part of the data stream and carry a timestamp t. A Watermark(t) declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with a timestamp t' <= t (i.e. events with timestamps older or equal to the watermark).

The figure below shows a stream of events with (logical) timestamps, and watermarks flowing inline. In this example the events are in order (with respect to their timestamps), meaning that the watermarks are simply periodic markers in the stream



Watermark (Contd.)

Watermarks are crucial for out-of-order streams, as illustrated below, where the events are not ordered by their timestamps. In general a watermark is a declaration that by that point in the stream, all events up to a certain timestamp should have arrived. Once a watermark reaches an operator, the operator can advance its internal event time clock to the value of the watermark.



Watermarks in Parallel Streams

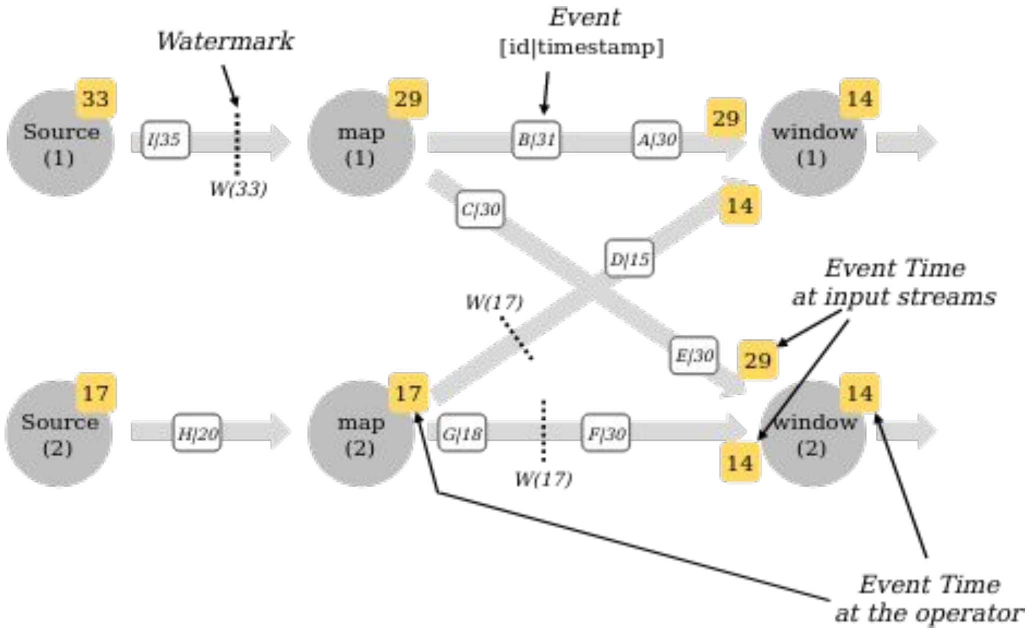
Watermarks are generated at, or directly after, source functions. Each parallel subtask of a source function usually generates its watermarks independently. These watermarks define the event time at that particular parallel source.

As the watermarks flow through the streaming program, they advance the event time at the operators where they arrive. Whenever an operator advances its event time, it generates a new watermark downstream for its successor operators.

Some operators consume multiple input streams; a union, for example, or operators following a *keyBy(…)* or *partition(…)* function. Such an operator’s current event time is the minimum of its input streams’ event times. As its input streams update their event times, so does the operator.

Watermarks in Parallel Streams (Contd.)

An example of events and watermarks flowing through parallel streams, and operators tracking event time.



https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/time/

Lateness

It is possible that certain elements will violate the watermark condition, meaning that even after the Watermark(t) has occurred, more elements with timestamp t’ <= t will occur.

In fact, in many real world setups, certain elements can be arbitrarily delayed, making it impossible to specify a time by which all elements of a certain event timestamp will have occurred. Furthermore, even if the lateness can be bounded, delaying the watermarks by too much is often not desirable, because it causes too much delay in the evaluation of event time windows.

For this reason, streaming programs may explicitly expect some late elements. Late elements are elements that arrive after the system’s event time clock (as signaled by the watermarks) has already passed the time of the late element’s timestamp.

Guarantees

- There are three well known message delivery guarantees associated with data stream processing
  - At-most-once
  - At-least-once
  - Exactly once
- When choosing the guarantee it is also important to remember that the performance will change. A stronger guarantee will impact the performance, or a weaker guarantee will increase the performance. E.g., to ensure "exactly once guarantee", additional work is required.

At-most-once delivery

- This guarantee will deliver the "best performance" and the messages will be processed once, or not at all.
- In this case, it is acceptable to lose messages, rather than deliver a message twice.

At-least-once delivery

- Records are never lost but may be redelivered.
- If your stream processing application fails, no data records are lost and fail to be processed, but some data records may be re-read and therefore re-processed.

Exactly-once delivery

- Records are processed once.
- Can assure us that every message is processed once and only once.
- The strongest guarantee available with stream processors
- Most difficult to implement

Challenges faced by data stream processing systems

- **Consistency** - historically streaming systems were created to decrease latency and made many sacrifices (e.g., at-most-once processing)
- **Throughput vs latency** - typically a trade-off
- **Time** - new challenge

Out-of-order event processing

- Out-of-order event arrival is present in general data stream processing applications.
- Out-of-order events in a stream are produced due to multiple reasons such as operator parallelization, network latency, merging of asynchronous streams, etc.
- Handling the disorder consists of a trade-off between result latency and result accuracy.
- There are four main techniques of disorder handling:
  - Buffer-based
  - Punctuation-based,
  - Speculation-based
  - Approximation-based.

Out-of-order event processing

- Buffer-based
  - These techniques sort tuples from the input stream using a buffer before presenting them to the query operator. K-Slack and AQ-K-slack are two example techniques for Buffer-based disorder handling. Buffer based techniques tolerate a specified amount of “slack” which is basically bound on the degree to which input could be unordered. Buffer-based mechanisms basically delay the processing for a specified slack period (e.g., T). Incoming data is
  - buffered for T time period and reordering of any out-of-order events which arrive within that period is conducted before any intelligent processing on the data has been conducted.

Out-of-order event processing

- Punctuation-based
  - Punctuation-based techniques depend on distinct tuples called Punctuations. These tuples orchestrates returning results pertaining to windows.
  - Hence unlike Buffer-based techniques, out-of-order events can be directly processed by the query.
- Speculation-based
  - These techniques operate as of no out-of-order events appear in the stream. When a window gets closed it produces the results immediately. Already produced results that get influenced by a late arrived event e are thrown away. After this invalidation the results are revised by considering the newly arrived e.
- Approximation-based
  - Approximation-based techniques make summaries of the data using some data structure such as q-digests, histograms, etc. They create rough results using the summary information.

Thank you!