Time and Order in Streams

NOTE: This document is a design draft and does not reflect the state of a stable release, but the work in progress on the current master, and future considerations.

Before implementing operations that rely in some way on time it is necessary to define the different concepts of time in Flink and how time is handled in a streaming topology.

Different Notions of Time

Some operations, such as time-windows and ordering by time, require a notion of time. In these operations the user has the choice between two approaches:

1. Using the current system time of the machine that an operation is running on, or
2. Using the timestamp of the records. In Flink, every element has a timestamp attached to it. When a value enters a streaming topology through a source, this source attaches a timestamp to the value. The timestamp can either be the current system time of the source (ingress time) or it can be a timestamp that is extracted from the value (event time).

Because of these two ways of assigning timestamps at sources, there are now three different types of times. All of which have different pros and cons:

- **Operator Time**: Operators simply trigger windows on their own system clocks. This usually gives the fastest results, since using timestamps in processing might require waiting for slower operations and buffering of elements since the elements will arrive at operations out-of-order. That is, they are usually not ordered by their timestamps. This will, however, not give you any guarantees about the ordering of elements when joining several data streams and when operations require a repartitioning of the elements since the timestamp of an element is essentially the time when it arrives at the operation.

- **Event Time**: This makes it possible to define orderings or time windows that are completely independent of the system time of the machines that a topology is running on. This might, however, lead to very high latencies because operations have to wait until they can be sure that they have seen all elements up to a certain timestamp.

- **Ingress Time**: This gives you a good compromise between event time and operator time. Even after repartitioning data or joining several data streams the elements can still be processed according to the time that they entered the system. The latency should also be low because the timestamps that sources attach to elements are steadily increasing.

Which time mode should be used is a program-wide setting, defined on the execution environment. The following reasons lead to this suggestion:

- Mixing timestamp time (event or ingress) and operator time leads to arbitrary and semantically ill-defined results.
- Switching between event time and ingress time should be simple. That way, we allow people to experiment with event time, and switch to ingress time, when they observe that event time latency is too high.
- Mixing ingress time and event time: This would probably be less interesting, as it gives again tricky results. If strictly needed (because some sources cannot assign an event time), one can use an "ingress time event time assigner".

Note: The implementation of Ingress Time and Event Time does not differ in any point but the data sources. All downstream operators are agnostic to where the timestamps come from.

Watermarks: Tracking the Progress of Time

This only applies to operations that use timestamp time (ingress time or event time) since operator time is only local to one parallel instance of an operation.

When using element timestamps to order elements or perform windowing, it becomes necessary to keep track of the progress of time. For example, in a windowing operator the window from \( t \) to \( t + 5 \) can only be processed once the operator is certain that no more elements with a timestamp lower than \( t + 5 \) will arrive from any of its input streams. Flink uses so called watermarks to keep track of the timestamp of tuples passing through the system: when a source knows that no elements with a timestamp lower than \( t + 1 \) will be emitted in the future it will emit a watermark with timestamp \( t + 1 \). Watermarks are broadcast to downstream operators. The watermark at one specific operator at any point in time is the minimum watermark time over all its inputs. Once that watermark increases the operator can react to that and further broadcast the watermark to downstream operators. In our windowing example from earlier, the operator would wait for the watermark for time \( t + 5 \), then process the window and then broadcast watermark \( t + 5 \) itself.

Since an operator has to wait for watermarks from all its inputs and also since timestamps can be very late compared to the current system time this can lead to latency (also called time skew or watermark skew). This figure should help understanding the problem:
The ideal case would be if the timestamp of the elements matched the processing time when they arrive at an operator. This rarely happens, though, and it can also happen that elements with an earlier timestamp arrive after elements with a later timestamp. The difference between the ideal watermark and the actual watermark at any point in time is the latency (or time skew).


### Generating Watermarks

To generate the watermark events at the data sources, three mechanisms are available:

1. **On Ingress Time**, every source generates the timestamps naturally in ascending order. Consequently, it can auto-generate watermarks fast, at a configurable rate (say every 10ms for 1:1 downstream connections, every 100ms for n:m downstream connections).

2. **On Event Time**, the user can specify that the data arrives with monotonously increasing timestamps per source partition. The source can then auto-generate watermarks (say at every 1s).

3. **On Event Time**, in the general case, the user needs to specify a watermark generator.

### Ordering Elements in a Stream

Elements in a stream can be ordered by timestamp. Normally, it would not be possible to order elements of an infinite stream. By using watermarks, however, Flink can order elements in a stream. The ordering operator has to buffer all elements it receives. Then, when it receives a watermark it can sort all elements that have a timestamp that is lower than the watermark and emit them in the sorted order. This is correct because the watermark signals that not more elements can arrive that would be intermixed with the sorted elements. The following code example and diagram illustrate this idea.
1. The time that closed the window
   - A meaningful option for time windows, where the result reflects the time it would have been computed under optimal watermark.
   - For time windows, elements retain their correspondence to a time slice.
   - Has the same meaning in operator time and ingress/event time.
   - For time windows, emits monotonous timestamps without extra effort.
   - For sliding windows followed by tumbling windows, the aggregated events align to the end in the new windows.

2. The time that opened the window
   - For time windows, elements retain their correspondence to a time slice.
   - Same meaning in operator time and ingress/event time.
   - No meaning with respect to time when computation happened.

3. The timestamp of the latest record in the window
   - For time windows, elements retain their correspondence to a time slice.
   - Meaningful for count windows: It reflects the time when the count window would have been computed under optimal watermarks.
   - Same meaning under operator time and ingress/event time.
   - For time windows, no meaning with respect to time when computation happened.

4. The timestamp of the earliest record in the window
   - For time windows, elements retain their correspondence to a time slice.
   - No meaning with respect to time when computation happened.

5. The watermark time or operator time of the operator when computing the group transformation.
5. Simple to implement, would guarantee ascending timestamps in the results.
   - Elements lose their correspondence to a certain time slice.
   - Meaning is different for operator time and event/ingress time.

The first two options can be views as folding the elements into a new element that is part of that time window.

Options (3) and (4) can be viewed as folding the elements into one of the window contents elements.

**Suggestion (by Stephan):**

Let us make the timestamp assignment configurable:

- Time windows (and session windows) use by default variant (1)
- Count windows use by default variant (3)
- Other windows use by default variant (3)

### Semantics of Sequences of Windows

Timestamp-based time windows are aligned on a fixed granularity, all other types of windows are not. We will see how this plays out and why it might be important in the following.

When you have several window operations in sequence you want to know which logical window an element belongs to at each state of processing. For example, in this code:

```scala
DataStream Example
val stream: DataStream[(Int, Int, Int)] = env.createStream(new KafkaSource(...))
val windowed: DataStream[(Int, Int, Int)] = stream
  .keyBy(0)
  .window(TimestampTime.of(5, SECONDS))
  .sum(2)
  .keyBy(1)
  .window(TimestampTime.of(5, SECONDS))
  .min(2)
```

we have two 5-second windows but the key by which the operations work is different. Since the timestamp for elements that result from the first window operations is the minimum of the timestamps inside the window the resulting element would also be in that same window. Since we have windows based on timestamps the windows are aligned. Therefore, the result of a window from \( t \) to \( t + 5 \) will also be in the window from \( t \) to \( t + 5 \) in the second window operation.

Technically speaking, if you have an element with timestamp \( t \) and windows of size \( \text{duration} \) (both in milliseconds) the start of the window that this element is in is given by \( t - (t \mod \text{duration}) \). The same applies to sliding time windows on timestamp time.

(This does not show how we re-partition the data after the first window operation, need to think about this picture.)
Accessing the Timestamp of an Element

Still TBD, this is a first idea.

If you have to access the timestamp of an element or change it you must use the special `flatMapWithTimestamp()` operation. This expects a `FlatMapWithTimestampFunction`:

```java
dataStreamExample {
    public interface FlatMapWithTimestampFunction<T, O> {
        void flatMap(T value, long timestampInMillis, TimestampCollector<O> out);
    }
    public interface TimestampCollector<O> {
        public void collect(O value, long timestampInMillis);
    }
}
```

The timestamp of the emitted element must not be lower than the timestamp of the input element. Otherwise the guarantees given by the watermarks would no longer hold.

Implementing Windows on Ingress- and Event Time

Still TBD, I have a proof-of-concept for time windows (aljoscha)

**Time-windows**

Pane based approach

**Other windows**

Staging buffer